(4)

A Multiprocessor Implementation Of CSP

by

Hwa-chung Feng

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

March 1988

Abstract

Communicating Sequential Processes (CSP) is a paradigm for communication and synchronization among distributed processes. The alternative construct is a key feature of CSP which allows *nondeterministic selection* of one among several possible communicants. Previous algorithms for this construct assume a message passing architecture and are not appropriate for multiprocessor systems which feature shared memory. The first part of this thesis describes a distributed algorithm for the alternative construct which exploits the capabilities of a parallel computer with shared memory. The algorithm assumes a generalized version of Hoare's original alternative construct which allows output commands to be included in guards. A correctness proof of the proposed algorithm is presented to show that the algorithm conforms to some *safety* and *liveness* criteria. Extensions to allow termination of processes and to explore fairness in guard selection are given. The second part of this thesis reports an implementation of the algorithm as part of a CSP system on the BBN Butterfly Parallel Processor™. The performance is measured and a method to fine-tune it is illustrated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The CSP Model

Communicating Sequential Processes (CSP) is a well known paradigm for communication and synchronization of a parallel computation [17,18]. A CSP program consists of a collection of processes P_1, P_2, ..., P_N that interact by exchanging *messages*. These message passing primitives, called input and output commands, are synchronous — a process attempting to output (input) a message to (from) another process must wait until the second process has executed the corresponding input (output) primitive.

An important feature of CSP is the *alternative* construct which is based on Dijkstra's guarded command [9]. This construct enables a process to *nondeterministically* select one communicant among many. Each alternative operation specifies a list of guards that identify the individual communicants. Each guard has a set of actions associated with it which cannot be executed until the value of the corresponding guard becomes TRUE. Each guard consists of a sequence of boolean expressions and an optional input command (output guards were not allowed in the original specification of CSP). A guard is said to be *enabled* if each of the boolean expressions preceding the input command evaluates to TRUE. The value of a guard is TRUE if the guard is enabled and its input action has successfully completed.

An alternative operation can be executed repeatedly in a *repetitive* construct which terminates when all the guards in the alternative operation have been disabled. A guard can be considered as "disabled" when the communicant identified by the guard terminates. Therefore, the termination of the repetitive construct can be caused by a combination of explicit disabling actions of the local process and termination of its communicants. This property is referred to as the *automatic termination* of the repetitive commands in [18].

1.2 Previous Works

Implementation of the alternative construct on a multiple processor computer has been the subject of much research [1,3,5,7,8,21,24,34]. It has been argued that the exclusion of output guards in the original definition of CSP is too restrictive. Examples in [5,11,24] show the lack of output guards forces the user to program unnecessary communications in situations where the action of an output guard is really needed, resulting in degradation of performance as well as an awkward program. The importance of output guard has been considered and treated by many researchers. Automatic termination is also important. It provides a clean semantics for terminating the *repetitive* construct. Lack of automatic termination may require the user to develop a complicated protocol between processes in order to determine the termination condition, which again can lead to subtle errors and awkward programs.

Algorithms that allow output guards in the alternative construct have been proposed[1,3,7]. Others suggest schemes that impose restrictions on possible communicants[36] or do not place upper bound on the time a process may spend in reaching communication agreements with its communicants[5,33]. However, none of them address the automatic termination problem. In fact, to the author's knowledge, none of these algorithms have ever been implemented and evaluated on a real machine. Occam [21] is a CSP based language developed for the Inmos

Transputer[20,30]. A very efficient implementation utilizing hardware support has been developed for it. At present, it is the only commercially available CSP system. However, it does not provide as primitives output guards or automatic termination.

1.3 Motivation Of Research

Most of the algorithms mentioned in the last section assume a message-based computer architecture; no shared memory is assumed. This is natural because CSP does not assume shared memory between constituent processes. One might ask why implementation of CSP on a shared memory machine is an issue. It is important for several reasons:

- CSP has clean semantics which simplifies proving the correctness of programs. It is a worthwhile programming paradigm in its own right, independent of the underlying machine architecture.

- The message passing paradigm is a natural means of expressing programs in many applications areas that are well suitable for shared memory machines. For example, distributed discrete event simulation algorithms are usually described in terms of message passing paradigms [22,26], and implementations on shared memory architectures have been described [32]. Similarly, message passing is used extensively in object-oriented programming.

- Shared memory machines are widely available. Multiprocessors such as the BBN Butterfly™ [2] and Sequent Balance™ are available from the commercial sector, and numerous shared memory research machines such as IBM's RP3 [15] and the University of Illinois's Cedar [16] have also been developed.

- Shared memory architectures provide fast interprocessor communications. A complete interconnection among processors is provided, avoiding costly store-and-forward communication software in message-based architectures such as

the Intel iPSC™ [31]. At present, multiprocessors are more appropriate for applications requiring frequent communication among the constituent processes.

Although one can clearly "retrofit" any message-based algorithm to a shared memory architecture by building a suitable interface, this will often lead to an inappropriate and awkward implementation. Existing message-based algorithms for the alternative construct are not appropriate for a shared memory machine because (1) they do not exploit the facilities afforded by shared memory, leading to an inefficient implementation; and (2) they require additional "system" processes to respond to incoming messages (e.g., requests for rendezvous) resulting in unnecessary context switching overhead. We will describe an algorithm for the CSP alternative construct that exploits the facilities afforded by shared memory and avoids the aforementioned system processes. This algorithm implements the "generalized" alternative construct that allows output guards, and also handles the automatic termination of repetitive commands.

The proposed algorithm uses the notion of total ordering among processes [5] to prevent deadlock, but applies this principle *dynamically* on transactions (defined later) rather than statically as originally proposed. The shared memory architecture simplifies the task of maintaining globally unique IDs. The status of a remote process can be interrogated directly, in contrast to the message-based algorithms where message handshake and context switching overheads reduce the efficiency of the implementation. However, because processes in the proposed algorithm concurrently access shared data, great care must be taken to avoid race conditions. Therefore, we provide a proof of the correctness of the algorithm according to *safety* and *liveness* criteria [23]. Modifications are also suggested to achieve fairness [14].

Finally, the algorithm does not contain any inherent communication *hot spots* [29]. Only one global variable is required, and is not accessed with sufficient fre-

quency to constitute a hot spot. With the exception of this variable, the algorithm is fully distributed and does not rely on any centralized controller.

1.4 Thesis Work

The principal contributions of this thesis are (1) to present an algorithm for implementing the *generalized* alternative construct on a shared memory multiprocessor which handles both output guards and automatic termination, (2) to prove its correctness in the sense of *safety* and *liveness*, (3) to realize a CSP system based on the proposed algorithm on a shared memory multiprocessor testbed, and (4) to evaluate the performance of the system. The thesis is organized as follows.

In Chapter 2, semantics of the generalized alternative construct is first discussed, followed by a description of the machine architecture on which it is based. The proposed algorithm and a discussion of its operation are presented next as well as other important issues related to the algorithm.

Chapter 3 develops a proof of the correctness of the algorithm according to some *safety* and *liveness* criteria. The fairness of the algorithm is also discussed.

Chapter 4 describes an implementation of CSP on the Butterfly Parallel Processor in detail. A brief introduction to the Butterfly architecture is presented, followed by the CSP language interface and the internal workings of the implementation.

Chapter 5 discusses the performance results of the Butterfly implementation and describes work to fine tune the alternative algorithm to maximize performance.

Chapter 6 summarizes the important results that have been achieved and suggests some future enhancements.

Chapter 2

The Alternative Algorithm

2.1 The Alternative Construct

A guard of the alternative construct can appear in one of two possible forms. The first, called the *pure boolean* form, contains no I/O command. For example, in

$$(x = 1 \; and \; y > 5) \rightarrow z := z * 3;$$

the predicate to the left of the '$\rightarrow$' operator is a pure boolean guard. The second form, called the *I/O guard* form, contains an I/O command as well as an (optional) boolean part. For example, in

$$P_1?x \rightarrow z := z + 1;$$

the input guard $P_1?x$ requests input from process P_1. The received data is assigned to the variable x. Guards such as this which do not contain a boolean part are referred to as *pure* I/O guards. In effect, the boolean part is the constant TRUE. An I/O guard is said to be *enabled* if the boolean part is TRUE, so a pure I/O guard is *permanently* enabled.

Consider the following alternative construct:

$$[G_{i(i \in PB)} \rightarrow S_i \; \square \; G_{j(j \in IO)} \rightarrow S_j].$$

Where PB stands for the set of indices of all of the pure boolean guards and IO the set of indices of all of the I/O guards. Whenever this alternative construct is

executed, exactly one guard is selected and the corresponding action (S_i or S_j) is executed. The selection is made according to the *availability* of the guards. For pure boolean guards, the guard is said to be available if it is enabled, i.e., if the boolean value evaluates to TRUE. For I/O guards, the guard is available if it is enabled and the process associated with the guard is also ready to communicate using the complementary I/O command. Because we assume I/O commands only appear in guards of alternative operations, this implies the remote process is executing an alternative operation in which the corresponding I/O operation is part of an enabled guard. If more than one guard is available, one is chosen arbitrarily. The application program cannot control this selection.

Pure boolean guards can be resolved without any interaction with other processes. Therefore, to simplify the discussion which follows, we will restrict attention to the resolution of I/O guards.

2.2 The Machine Architecture

The machine is assumed to be a shared memory multiprocessor. The algorithm is well suited for machines such as BBN's Butterfly or Sequent's Balance, among others. Several primitives are used in the algorithm. None are unusual in a multiprocessor environment, and all can be easily constructed using a test-and-set and standard scheduling primitives.

The CSP program contains processes P_1, P_2, ..., P_N. Process P_i is assigned the unique *process ID* i to distinguish it from others.

We will assume the following:

- Communications are reliable. An error free communication mechanism exists so that two distinct processes can communicate by exchanging a message. In particular, **Send(M, R)** and **Recv(R): Message** provide the same semantics as CSP's output and input commands, respectively. M is the message

which is transmitted and R is the ID of the remote process with which communications is to take place. *Recv* returns the received message (of type *Message*). In accordance with CSP semantics, we assume the process invoking the primitive blocks until process P_R executes the complementary I/O primitive.

- Read and write accesses to shared memory are atomic, as is normally the case with a shared memory multiprocessor. **AtomicAdd(X): INTEGER** atomically increments the integer variable X and returns the original value of X.

- *WaitForSignal* and *Signal* primitives are available to block and unblock the process, respectively. A signal contains a single, user defined integer value. **WaitForSignal(): INTEGER** causes the process invoking the primitive to block until a signal becomes available to it from *any* other process and returns the integer value stored within the signal. **Signal(R, i)** sends a signal containing integer i to process P_R. The *Signal* primitive wakes up the signaled process if it is blocked on *WaitForSignal*. Otherwise, the signal remains in effect until P_R executes a *WaitForSignal* primitive. If a second signal is sent to P_R before the first is received, the first signal is discarded.

- *Lock* and *Unlock* primitives provide exclusive access to shared data structures. **Lock (L)** will block until the lock L becomes zero, at which time L is set to one. The "test-and-set" operation must be atomic. **Unlock (L)** sets the lock L to one. Further, we assume the *Lock* primitive is fair, i.e., if a process is blocked while attempting to obtain a lock, it does not remain blocked for an unbounded amount of time unless the lock is not unlocked for an unbounded amount of time.

- **Sleep(T)** causes the process invoking it to block for at least T time units. A

process will always be eventually awoken after calling *Sleep*.

- The amount of time between memory reads to a shared memory location by successive machine instructions of a program executing on a single processor can be bounded. This may require disabling interrupts for a short period of time.

The final assumption listed above is the strongest requirement of the proposed algorithm. It is not necessary to ensure the safety of the algorithm, i.e., if it were relaxed, no "invalid" rendezvous will result. However, it is necessary to prove the liveness of the algorithm. In particular, it is necessary to avoid an, albeit unlikely, scenario in which one process P_i is in a "busy wait" loop polling a variable of another process P_j, and P_j (1) modifies the variable, (2) invokes the *Sleep* primitive, and then after awakening (3) restores the variable to its original value before step (1). All of this must occur *without* P_i noticing the variable had been modified, i.e., these events must occur *between* sucessive samples of the variable by P_i's polling loop. Further, this scenario must repeat an unbounded number of times in succession to compromise the liveness of the algorithm. Therefore, it is a rather mild requirement of the proposed algorithm that can be relaxed in practical situtations.

It is assumed that all input and output commands occur within guards of the alternative construct. Simple CSP input and output primitives are special cases of the alternative construct. We also assume that the variables used in the alternative algorithm are not modified by processes except as indicated in the algorithm. Finally, it is assumed that processes do not terminate. The algorithm can be extended to handle termination, as will be discussed later.

2.3 The Alternative Algorithm

Each invocation of an alternative operation is referred to as a *transaction*. A transaction begins when an alternative operation is initiated and ends when a suc-

cessful communication has been completed. A process will usually engage in many transactions during its execution. A total ordering is imposed among all transactions entered by *all* processes of a given CSP program. A unique sequence number, referred to here as a *transaction ID*, is associated with each transaction.

Two processes which each initiate an alternative operation that results in a communication between them are said to *rendezvous*. More precise definitions of rendezvous and other terminology introduced in this section will be presented later. Each rendezvous always involves exactly two distinct processes. In a *typical* rendezvous, the first process to enter the alternative will block, waiting for a signal from the second. When the second process enters the alternative, it will *commit* to the first in order to obtain "permission" to rendezvous; the "committing" process will then signal and exchange a message with the blocked process, and both will complete their respective alternative operations.

A *commit* operation is, in effect, a request for rendezvous. It will be shown that a rendezvous will occur only after a successful commit operation has taken place, and every successful commit results in a rendezvous. A process will not attempt to commit until it has determined that the process with which it is committing is a suitable candidate for rendezvous, i.e., each lists the other in their respective guard lists, and the two processes are not both trying to execute the *same* I/O operation (send or receive). The commit operation resolves conflicts when two different processes attempt to simultaneously rendezvous with a third. The algorithm uses an "abort/retry" mechanism to avoid race conditions when two potential communicants simultaneously enter the alternative command.

2.3.1 Process States

Each process can be in one of the following states:

- **WAITING.** The process is blocked on a *WaitForSignal* operation, waiting for another process to rendezvous with it.

- **ALT.** The process has begun an alternative operation, and is scanning through its list of guards to find a process with which it can rendezvous.

- **SLEEPING.** The process was forced to abort an alternative operation. After aborting, the process goes to sleep for some predetermined period of time before retrying. While blocked in this way, the process is in the SLEEPING state. This state differs from the WAITING state because a process may remain in the latter for an unbounded amount of time.

- **RUNNING.** The process is executing user or system code not related to the alternative operation. The process is in the RUNNING state if it is not in any of the other states listed above. Once the process initiates an alternative operation it can only be in the WAITING, ALT, or SLEEPING state until the alternative operation completes with a rendezvous.

It is possible to combine the RUNNING and SLEEPING states into a single state. Two states are used to simplify the description of the algorithm and its proof.

A state transition diagram for each process is shown in figure 2.1. Initially, a process is in the RUNNING state. Once the process initiates an alternative operation, it enters the ALT state. If the process is forced to abort the alternative it switches to the SLEEPING state, and returns to the ALT state when it retries. If the process is able to commit and rendezvous with another process, it returns to the RUNNING state. Otherwise, the process moves to the WAITING state until some other process commits to it, at which time it rendezvous and returns to the RUNNING state.

The ALT and SLEEPING states should be viewed as "transitory" states through which a process must pass while trying to commit or move into the WAITING state.

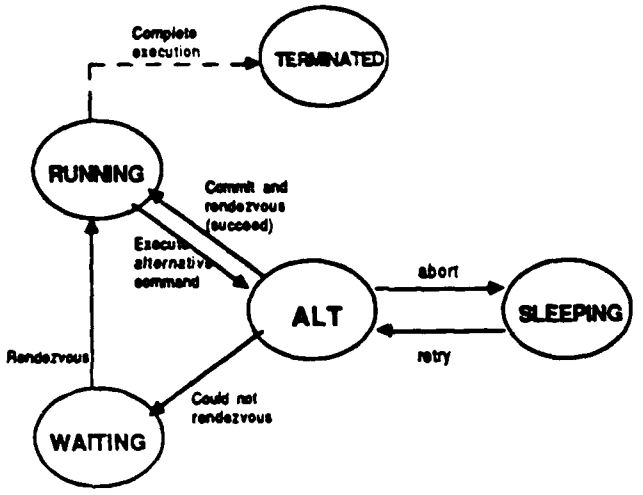

Figure 2.1: The State Diagram of a Process

It will be shown that a process cannot remain in either the ALT or the SLEEPING state for an unbounded amount of time on a single transaction.

2.3.2 Shared Variables

Each process P_j maintains a number of variables which may be examined, and in some cases modified, by other processes:

- **AltList$_j$** lists the guards associated with the last alternative operation initiated by P_j that caused P_j to enter the WAITING state.
- **AltLock$_j$** is a lock used to control access to $AltList_j$. It is initialized to 0 (unlocked).
- **State$_j$** holds the current state of P_j. It may be set to WAITING, ALT, SLEEPING, or RUNNING, and is initialized to RUNNING.
- **WakeUp$_j$** is initialized to 1 and is set to zero by P_j whenever it enters the WAITING state. It is incremented (atomically) by processes trying to commit

to P_j. This variable prevents two processes from both successfully committing to a third on a single transaction.

There is also one system wide global variable used by the algorithm:

- **NextTransID** is initialized to zero and is incremented each time a process initiates an alternative operation. This variable ensures a unique transaction ID can be generated for each instance of an alternative operation.

One procedure merits special attention. **CheckAndCommit(AltList$_r$, g$_i$): INTEGER** is called by process P_l to check that "valid" communications can take place between P_l using guard g_i and P_r, and if so, to attempt to commit to P_r. If a commit was attempted and succeeded, then *CheckAndCommit* returns a positive integer indicating the corresponding guard in the *remote* process P_r. Otherwise, *CheckAndCommit* returns a non-positive integer, denoted by the constant FAILED. This procedure is shown in figure 2.2.

CheckAndCommit uses a procedure **CheckGuard(AltList$_r$, g$_i$): INTEGER** that scans the remote alternative list $AltList_r$ looking for a *matching* and *compatible* guard g_j to the local guard g_i. By *matching* we mean g_j contains an I/O operation with P_l. By *compatible* we mean g_i and g_j do not *both* contain input (output) commands. *CheckGuard* returns j, the number of a matching and compatible guard if one was found, and FAILED otherwise. If such a guard is found, P_l attempts to commit to P_r by testing if $WakeUp_r$ is zero, and if so, incrementing it. An ordinary addition is used rather than the *AtomicAdd* primitive to increment $WakeUp_r$ because $AltLock_r$ guarantees atomicity — every "test-and-set" operation performed on $WakeUp_r$ occurs while $AltLock_r$ is set. If P_l is the first process to commit to P_r, i.e., if $WakeUp_r$ was previously zero, then P_l successfully commits, *CheckAndCommit* returns the number of the corresponding guard, and rendezvous is imminent. Otherwise, *CheckAndCommit* returns FAILED. $AltLock_r$ ensures serial

```
/* r is the remote process */
PROCEDURE CheckAndCommit(r, g_i): INTEGER;
VAR
    INTEGER GuardNumber; /* number of matching guard */
BEGIN
    Lock (AltLock_r);
    /* check if guard matches and is compatible */
    GuardNumber := CheckGuard(AltList_r, g_i);
    IF (GuardNumber = FAILED) THEN
        Unlock (AltLock_r);
        RETURN (FAILED);
    /* try to commit */
    ELSEIF (WakeUp_r = 0) THEN
        WakeUp_r = WakeUp_r + 1;
        Unlock (AltLock_r);
        RETURN (GuardNumber);
    ELSE
        Unlock (AltLock_r);
        RETURN (FAILED);
        END;
END CheckAndCommit;
```

Figure 2.2: Procedure to check for potential communication and to commit.

access to $AltList_r$. As will be demonstrated later, it is crucial that this lock is not released until *after* the commit operation is attempted (if it is attempted) in order to avoid race conditions. This would be the case even if an *AtomicAdd* operation were used to increment the *WakeUp* variable.

2.3.3 Other Notation

For notational convenience, other variables and predefined functions are defined that are used in the algorithm. These include:

- **TransID$_l$** is a variable that contains the ID of the current transaction in which process P_l is engaged.
- **CommunicantID(g_i)** is a function that returns the ID of the process listed in the I/O command portion of guard g_i.
- **Communicate(g_i)** executes the I/O command in guard g_i.

```
/* g_i are enabled I/O guards */
PROCEDURE Alternative(g_1, ..., g_n): INTEGER;
VAR
    INTEGER ReturnValue; /* indicates guard which rendezvoused */
BEGIN
    /* 1 is the local process id */
    TransID_1 := AtomicAdd(NextTransID);
    ReturnValue := FAILED;
    WHILE (ReturnValue = FAILED) DO
        ReturnValue := TryAlternative (g_1, ..., g_n);
        IF (ReturnValue = FAILED) THEN Sleep (TimeOut); END;
        END;
    RETURN (ReturnValue);
END Alternative;
```

Figure 2.3: The "front end" procedure.

- **TimeOut** is a constant indicating the number of time units a process should sleep after an aborted attempt. More will be said about this later.

2.3.4 Description Of The Algorithm

The alternative algorithm is shown in figures 2.3 and 2.4. The *Alternative* procedure shown in figure 2.3 is a "front end" which is responsible for retrying aborted attempts. The heart of the algorithm lies in the *TryAlternative* procedure shown in figure 2.4. The parameters passed to both procedures are n enabled I/O guards $g_1, g_2, \ldots, g_n$. Each guard contains either a single output or a single input primitive. The *Alternative* procedure is only called after non I/O guards have been evaluated and are found to be FALSE. This procedure does not return until a rendezvous has been completed, at which time it returns an integer indicating the guard ($g_1, g_2, \ldots, g_n$) that was eventually satisfied.

The *Alternative* procedure obtains a unique transaction ID by performing an *AtomicAdd* operation on the global *NextTransID* variable. It then attempts to rendezvous by calling *TryAlternative*. *TryAlternative* either returns the number of the guard on which a rendezvous occurred, or the FAILED flag indicating the

```
PROCEDURE TryAlternative(g1, ..., gn): INTEGER;
VAR
    BOOLEAN flag;
    INTEGER GuardNumber; /* corresponding guard of Pr */
    INTEGER i, r;
BEGIN
    State1 := ALT;
    /* look for rendezvous with a waiting process.  */
    FOR i:=1 TO n DO
        r := CommunicantID(gi);
        flag := TRUE;
        WHILE (flag) DO
            CASE Stater DO /* The remote process state.  */
            SLEEPING: flag := FALSE;
            RUNNING: flag := FALSE; /* try next guard */
            WAITING: GuardNumber := CheckAndCommit(r, gi);
                IF (GuardNumber = FAILED) THEN
                    flag := FALSE; /* try next guard */
                ELSE /* Wake up Pr */
                    State1 := RUNNING;
                    Signal(r, GuardNumber);
                    Communicate(gi);
                    RETURN (i);
                END;
            ALT: IF (TransID1 > TransIDr) THEN
                    State1 := SLEEPING;
                    RETURN (FAILED); /* abort...*/
                ELSE /* busy wait loop.  */
                    WHILE ((Stater = ALT) DO END
                END; /* if-then-else */
            END; /* case statement */
        END; /* while loop */
    END; /* for statement */
    /* couldn't find guard to rendezvous */
    Lock(AltList1); AltList1 := (g1, ..., gn); Unlock(AltList1);
    WakeUp1 := 0; /* first to commit gets rendezvous */
    State1 := WAITING;
    i := WaitForSignal(); /* Blocks */
    State1 := RUNNING;
    Communicate(gi)
    RETURN (i);
END TryAlternative;
```

Figure 2.4: *TryAlternative* attempts rendezvous with communicants.

attempt must be retried. Each time *TryAlternative* fails, the process enters the SLEEPING state for at least *TimeOut* time units before retrying. The same transaction ID remains in use despite one of more failed attempts. It will be shown that *TryAlternative* cannot fail an unbounded number of times within a single transaction.

The heart of the alternative algorithm is embodied in the *TryAlternative* procedure (figure 2.4). In this procedure, l refers to the local process P_l, and r refers to the remote process P_r associated with the guard that is being scanned.

After setting the state of the process to ALT, P_l examines each guard listed in the alternative operation one after the other. Some action is then performed depending on the state of P_r.

If P_r is in the RUNNING state, P_l simply advances to the next guard. In this case, P_r has not yet entered a transaction and is not yet ready to rendezvous.

If P_r is in the SLEEPING state, P_l again advances to the next guard. P_l advances because the *Alternative* procedure guarantees that the SLEEPING process (P_r) will eventually retry its alternative operation. If P_l and P_r are destined to eventually rendezvous on this transaction, P_l will typically proceed to the WAITING state, and P_r will later retry, commit, and rendezvous with P_l.

If P_r is WAITING, then P_r has already reached the rendezvous point so P_l attempts to rendezvous. $AltList_r$ is examined to make sure a valid communication can take place, and if so, P_l attempts to commit. If successful, P_l will awaken P_r (by sending a signal) and rendezvous. Otherwise, P_l advances to the next guard.

Finally, if P_r is in the ALT state, some special precautions must be taken to avoid race conditions. This situation could result, for example, when P_l and P_r initiate an alternative operation at approximately the same time. The two processes may or may not be destined to rendezvous, however. In fact, P_r's alternative operation may not even contain a guard with P_l as a communicant.

If two processes see each other in the ALT state, one will be forced to abort and retry the alternative, while the other pauses within the current operation until the first aborts. The transaction IDs of the two processes are used to determine the process that will abort and the process that will proceed. A process with a smaller, i.e., older, transaction ID is given higher priority. This protocol avoids deadlock situations in which two processes attempting to communicate with each other both advance to the WAITING state.

If the process does not abort, it pauses in a busy wait loop until the remote process moves out of the ALT state. The remote process will either abort, changing to the SLEEPING state, or rendezvous, changing to the RUNNING state. Later, it will be shown that one of these two possibilities must eventually occur. Although the busy wait loop and abort retry scenario might initially appear to cause wasted time that could be better spent pursuing other activities, it is anticipated that this situation will arise infrequently in practice. Performance evaluations using empirical techniques will be described in a later chapter.

It is interesting to note that the state of P_r may change immediately after P_l examines $State_r$. It will be proven that the algorithm operates correctly despite this potential inconsistency. In fact, it will be shown that the only locking that must be performed in the entire algorithm is that associated with *AltLock*.

If P_l goes through its entire guard list without rendezvousing with another process, P_l enters the WAITING state and calls *WaitForSignal* to block until another process commits to it. Before calling *WaitForSignal*, however, P_l also sets $AltList_l$ to contain the current guard list and "activates" $WakeUp_l$ by setting it to zero. After some process later commits to P_l, a signal is received, a communication takes place, and *TryAlternative* returns the identity of the (local) guard that rendezvoused. This information is sent to P_l in the signal that awakened it.

We should emphasize at this point that it is crucial that the operations listed

in figures 2.2, 2.3, and 2.4 be performed in *exactly* the order in which they appear. Seemingly minor changes such as swapping the order of the statements

```
WakeUp[1] := 0;
State[1] := WAITING;
```

introduces a race condition that invalidates the correctness proof.

We note that the *Lock* operation preceding the statement that modifies *AltList* must remain even if modification can be done atomically. The locking protocol in this and the *CheckAndCommit* procedure are carefully designed to avoid race conditions. Finally, it is noteworthy that the statement that sets $WakeUp_l$ to zero need *not* be executed while $AltLock_l$ is locked. The correctness proof only requires that two processes do not both read a zero value from $WakeUp_l$ during a single transaction of P_l. This is guaranteed by the locking protocol used in *CheckAndCommit*.

2.4 Discussion

Several aspects of the alternative algorithm presented above merit further discussion. These will be discussed next.

2.4.1 Transaction IDs

The algorithm uses dynamically assigned transaction IDs to determine the "winner" when a process finds another in the ALT state. Dynamic IDs are used rather than static, process IDs to ensure liveness. Intuitively, *liveness* means that two processes that "should" rendezvous eventually will, while *safety* means that any rendezvous that occurs is valid. The proposed approach avoids scenarios in which a process is repeatedly forced to abort and retry its alternative operation an unbounded number of times; this is because the priority of a transaction automatically

increases with time as other transactions are allowed to complete and new ones, with higher IDs and correspondingly lower priorities, are initiated. Dynamic transaction IDs guarantee this property while static IDs do not. It is important that a new transaction ID is only allocated when an alternative is first initiated, as is done in figure 2.3, and *not* when an existing operation is retried. The use of dynamic transaction IDs is further justified by the fact that global variables are relatively inexpensive in shared memory architectures, and the *NextTransID* variable is not referenced with sufficient frequency to become a hot spot.

A second concern is overflow of the *NextTransID* variable. Overflow invalidates the liveness property of the algorithm because a transaction's priority does not necessarily increase with time. Also, because transaction IDs cannot be guaranteed to be unique after overflow has occurred, the arbitration protocol could fail (this could be circumvented by appending the process ID to the least significant portion of the transaction ID, however). In any event, overflow can be easily avoided by using a variable of large precision. For example, a 64 bit variable will not overflow with 1000 processes, each initiating a new alternative construct every microsecond, in over 500 years!

2.4.2 The Timing Assumption

We earlier required the following assumption to ensure liveness:

> The amount of time between successive samples of a shared memory location by a busy wait loop (which does nothing but sample and test the value stored in this location for inequality) can be bounded, and is shorter than the time required to invoke either the *Send* or *Recv* primitives.

This assumption is necessary because the algorithm uses a polling loop to detect another process leaving the ALT state. Suppose P_i is waiting for P_j to change to a

new state. It is possible, albeit unlikely, that P_j (1) modifies $State_j$, (2) rendezvous and resumes execution of user code or goes to sleep for *TimeOut* units of time, and (3) reenters *TryAlternative* and changes $State_j$ back to ALT; all of this must occur *without* P_i noticing $State_j$ had been modified, so this activity must occur *between* successive samples of $State_j$ by P_i's polling loop. While it is true that this might occasionally occur if P_i is interrupted during its polling loop, it is necessary that this scenario be repeated ***an unbounded number of times*** within a single execution of the polling loop to compromise the liveness of the algorithm. We conjecture that it is highly improbable that such a scenario will occur even a few times within a single transaction. Further, we emphasize that safety remains guaranteed even if the above assumption is relaxed, so no ill effects, other than delays, will result should this scenario occur some (finite) number of times.

As can be seen from figure 2.4, P_j must execute either the *Sleep*, *Send*, or *Recv* primitive *after* the state of P_j is changed (to SLEEPING or RUNNING), i.e., during step (2) above. Therefore, as stated in the above assumption, ensuring that the minimum execution time of each of these primitives exceeds the time between successive samples of P_i's polling loop is sufficient to avoid the above scenario (actually, the *Sleep* primitive is excluded because its minimum execution time is trivially set). If the time between successive samples of the polling loop can be bounded, the minimum amount of time required by the *Send* and *Recv* primitives can be easily modified to adhere to the timing assumption through the introduction of a timed delay (e.g., by calling *Sleep*). However, one would not expect introduction of such a delay to be necessary in most practical situations.

Assuming the time required by a remote memory reference is bounded, the time between successive samples by the busy wait loop can be bounded by disabling interrupts during the polling loop. If this is not a viable alternative, one can reduce the likelihood of entering the above scenario by introducing randomness into the

program's temporal behavior. For example, a random sleeping period may be selected (with some minimum value, as described below) when a process is forced to abort. This will reduce the likelihood of excessive delays caused by synchronized behavior between processes.

2.4.3 Setting the Sleeping Period

The "sleep period" before a retry is attempted, i.e., *TimeOut* in figure 2.4, must be sufficiently long to allow the "winning" process to observe that the sleeping process is indeed in the SLEEPING state. In particular, *TimeOut* cannot be shorter than the interval between successive samples in the busy wait loop executed by the winner. This is the reason we earlier assumed the time between successive samples could be bounded. Relaxing this constraint, e.g., to allow either process to be descheduled during this busy wait loop, could in principle jeopardize liveness (but not safety). However it is unlikely that unbounded abort and retry scenarios will occur in practical situations.

On the other hand, an excessively long sleeping period will lead to an inefficient implementation. A method of fine-tuning the sleeping period for better performance will be discussed in a later chapter.

2.4.4 Channel I/O

In many CSP implementations, interprocess communication is based on pre-allocated *channels*. Each channel is an unilateral link between two communicating processes. The channel model facilitates modularity, reusability, and hierarchical construction of programs since a program can be "constructed" by interconnecting a group of constituent processes. The algorithm presented above can be easily adapted to the channel I/O model by modifying the Send and Recv primitives and translating port identifiers to process IDs. The channel model is adopted in the

implementation described later.

2.4.5 Termination

Termination is another issue facing real implementations. This was not treated in the previous discussion because it complicates the proof and obscures the description. The termination semantics play an interesting role in CSP because it is the basis of the termination of the *repetitive* command [18]. However, it is not found in the semantics of Occam[21]. A repetitive construct terminates if in its enclosed alternative operation all of the guards are either *disabled* or contain an I/O operation with a terminated task.

The algorithm can be extended to handle termination by adding a shared variable called $GuardCount_i$ to each process P_i and a new process state called TERMINATED. $GuardCount_i$ indicates the number of I/O guards on which P_i might eventually rendezvous when P_i is in the WAITING state. It is set by P_i *before* P_i sets $State_i$ to WAITING. The $GuardCount_i$ variable is used to detect situations in which P_i cannot rendezvous because all of the processes in its guards have terminated. In this case, we say the alternative operation in P_i fails.

Whenever a process P_j terminates, it marks its state as TERMINATED and then examines the state of each of its neighboring processes, i.e., those processes which might communicate with P_j. If P_j finds another process P_i in the WAITING state and $AltList_i$ contains a guard listing P_j as a communicant, then P_j atomically decrements $GuardCount_i$ to indicate that one fewer guard is available for rendezvous. No further action is taken unless the decrement operation causes $GuardCount_i$ to become zero, i.e., the atomic decrement operation returned one, the value before the decrement. In this case, the terminating process must send P_i a special signal to indicate P_i's alternative operation can never rendezvous. Upon receiving this signal, P_i will return a special flag to the process indicating the al-

```
PROCEDURE Termination()
VAR
    BOOLEAN flag;
    INTEGER r;
BEGIN
    State_I := TERMINATED;
    FOR g_i ∈ Neighbors_I DO
        r := CommunicantID(g_i);
        flag := TRUE;
        WHILE (flag) DO
            CASE State_r DO /* The remote process state.  */
            TERMINATED: flag := FALSE;
            SLEEPING: flag := FALSE;
            RUNNING: flag := FALSE; /* never mind */
            ALT: WHILE ((State_r = ALT) DO END
            WAITING: CheckAndSignal(r, g_i);
                     flag := FALSE;
            END; /* case statement */
        END; /* while loop */
    END; /* for statement */
    EXIT (); /* Really terminate */
END Termination;
```

Figure 2.5: *Termination* procedure executed by terminating processes.

ternative operation completed *without* rendezvous. Figure 2.5 shows the algorithm that a process executes when it terminates. The variable $Neighbors_j$ represents the list of all guards that process P_j can have during its life time. *CheckAndSignal* is similar to *CheckAndCommit* and is shown in figure 2.6.

When scanning the status of neighboring processes in the *TryAlternative* procedure, an I/O guard corresponding to a terminated process is skipped in the same way processes in the RUNNING or SLEEPING state are skipped. If all I/O guards correspond to terminated processes, the alternative construct similarly returns a flag indicating the operation completed without rendezvous.

```
/* r is the remote process */
PROCEDURE CheckAndSignal(r, g_i)
VAR
     INTEGER GuardNumber; /* number of matching guard */
BEGIN
     Lock (AltLock_r);
     /* obtain guard number */
     GuardNumber := CheckGuard(AltList_r, g_i);
     IF (GuardNumber = FAILED) THEN
         Unlock (AltLock_r);
     ELSEIF (GuardCount_r = 1) THEN
         Unlock (AltLock_r);
         signal(r, SpecialSignal);
     ELSE
         Unlock (AltLock_r);
         END;
END CheckAndCommit;
```

Figure 2.6: Procedure to check for potential repetition exit and to signal.

Chapter 3

Proof Of Correctness Of The Algorithm

The correctness of the algorithm is established by proving that during the (potentially) infinite execution sequence, all processes and the interplay between them maintain invariant properties known as *safety* and *liveness* [23,28]. As described above, safety means that any rendezvous which occurs is correct. For example, it is not possible for two processes to rendezvous which do not each list the other in some guard of their respective alternative lists. Liveness ensures that two processes which should rendezvous eventually will, provided of course each does not first rendezvous with some other process. These terms are defined more formally in theorems 2 and 3. Intuitively, the safety property ensures that nothing "bad" will happen, while liveness ensures something "good" will eventually happen. Together they guarantee correct operation of the algorithm.

Before beginning the proof, terminology which has been used informally until now will be defined more precisely. These definitions are in terms of the alternative algorithm shown in figures 2.3 and 2.4. It is assumed throughout that the CSP program consists of a collection of processes, $P_1, P_2, \ldots, P_N$.

3.1 Definitions

1. A process P_i is said to enter a transaction T_r when it calls the *Alternative* function. It exits transaction T_r when it returns from the function call. $P_i(T_r)$ denotes that fact that P_i is in T_r. Each transaction has a unique ID associated

with it (r for transaction T_r) that is used to form a total ordering among all transactions. A transaction need not terminate. For example, the application program may contain deadlock situations.

2. A Process P_i in transaction T_r is said to *commit* to process P_j if $P_i(T_r)$ increments $WakeUP_j$ from zero to one.

3. A transaction T_r executed by process P_i is said to rendezvous with transaction T_s for process P_j if either (a) P_i is in the WAITING state and receives a signal from P_j, or (b) P_i signals P_j after committing to P_j. It will be shown that once a process rendezvous, it will exchange a message, complete the current transaction and return to the RUNNING state.

4. A signal sent by P_i to P_j is said to be *pending* if (1) it was sent but has not yet been received by P_j, or (2) if it was received, but has not yet been absorbed by P_j through a call to *WaitForSignal*.

5. A communication between P_i and P_j is *compatible* if one process wishes to send, and the other wishes to receive. Otherwise, the communication is said to be *incompatible*.

6. $VAR_i(T_r)$ denotes the value of state variable VAR of process P_i during transaction T_r. For example, $AltList_i(T_r)$ is the alternative list of P_i during transaction T_r. If significant, the point in time *during* the transaction that is referred to will be stated explicitly.

7. The function $prev(T_r)$ returns the ID of the transaction executed by the process which immediately preceded T_r. The existence of T_r implies the termination of prev(T_r). Also, $prev^0(T_r)$ refers to T_r itself and $prev^m(T_r)$ corresponds to the mth previous transaction entered by P_j.

8. **GuardList$_i$(T_r)** lists the guards that are passed as parameters to the alternative operation executed by P_j on transaction T_r. We will take the liberty of giving *GuardList* a dual meaning — it either refers to a list of *guards* or a list of *process* that are designated in the I/O commands of these guards. The particular meaning that is intended will be clear from the context.

3.2 The Safety Property

Lemmas 1 through 5 lead to theorem 1 which states that no race conditions arise that might cause a process to mistakenly rendezvous with a second process that does not wish to rendezvous with the first. Theorem 2 subsumes theorem 1 and ensures that the algorithm obeys the safety property.

Lemma 1 *$P_i(T_r)$ signals P_j iff $P_i(T_r)$ commits to P_j.*

> **Proof:** This follows immediately from examination of the algorithm. A process only sends a signal after it commits, and always sends a signal after it commits. ∎

This lemma implies that $WakeUp_j$ must be set to 0 before a signal can be sent to P_j. In addition, at most one signal is sent to P_j each time $WakeUp_j$ is set to 0.

Lemma 2 *At the beginning and at the end of each transaction entered by P_j, the following conditions must hold:*

(a) No signals sent to P_j are pending.

(b) WakeUp$_j$ *is nonzero.*

> **Proof:** Use induction on m, the number of transactions entered by P_j.
>
> Consider the first transaction ($m = 1$) executed by P_j. $WakeUp_j$ is initialized to 1. Because $WakeUp_j$ can only be set to 0 by P_j during a transaction, $WakeUp_j$ must remain nonzero up to at least the beginning

of P_j's first alternative operation. No process can commit to P_j until $WakeUp_j$ becomes 0, so by lemma 1, no signals can be sent to P_j before its first transaction, and therefore none can be pending. Thus, (a) and (b) are both true at the beginning of P_j's first transaction.

During any transaction, and in particular the first, P_j will either reset $WakeUp_j$ to 0 exactly once (just before entering the WAITING state), or not at all. If P_j does not reset $WakeUp_j$, then obviously $WakeUp_j$ is still nonzero at the end of the alternative operation. No signal can be sent to P_j because no process can commit, so none are pending.

If P_j does reset $WakeUp_j$ to 0, then at most one process can commit (and send a signal) to P_j during this transaction. This is because (1) $WakeUp_j$ is set to 0 at most one time during this transaction; (2) each process must obtain the lock $AltLock_j$ before it can examine $WakeUp_j$ (see the *CheckAndCommit* procedure); (3) as soon as one process reads a zero in $WakeUp_j$, it increments it *before* releasing $AltLock_j$; so (4) two processes cannot both read a zero value from $WakeUp_j$ during a single transaction in P_j. Because no two processes can see a zero value in $WakeUp_j$ during a single transaction, no two processes can commit to P_j during this (or any) transaction. Therefore, according to lemma 1, at most one signal will be sent to P_j during this transaction.

P_j always calls *WaitForSignal* after setting $WakeUp_j$ to zero. Therefore, the only signal that could have been sent to P_j must have been absorbed by the *WaitForSignal* operation, so none can be pending when the transaction completes (if it completes) satisfying condition (a). Condition (b) must also be satisfied at the end of the transaction because a process must commit *before* sending a signal to P_j, so $WakeUp_j$ must be nonzero

before the process can resume execution after calling *WaitForSignal*. Therefore, (a) and (b) are again true at the end of the first alternative operation as well as at the beginning.

Inductive step: Assume lemma 2 is true on the mth transaction entered by P_j. We will now show it is also true on the $m+1st$ transaction. According to the inductive hypothesis, no signals are pending at the end of the mth operation, and $WakeUp_j$ is nonzero. Therefore, these conditions will remain true until the beginning of the $m+1st$ transaction because no process can commit to P_j until $WakeUp_j$ becomes 0. As noted in the proof for $m = 1$, if (a) and (b) are true at the beginning of any transaction, they will be true at the end of the transaction if it terminates. Therefore, (a) and (b) are true at the end of the $m+1st$ transaction entered by P_j. ∎

Lemma 3 *Two processes, P_i and P_j, cannot both commit to a third process P_k during a single transaction T_t entered by P_k.*

This lemma was actually proven as part of the proof of lemma 2, but we include it as a separate lemma for future reference. The proof relies on the fact that $WakeUp_k$ is not zero at the beginning of the alternative operation and can be set to zero at most one time during a single transaction. The atomicity of the commit operation (i.e., two read-modify-write sequences cannot be inappropriately interleaved) guarantees that only a single process can commit to P_k during T_t.

Lemma 4 *If $P_i(T_r)$ commits to P_j, then P_j must have been in the* WAITING *state when P_i committed to P_j, and P_j must remain in the* WAITING *state until P_j receives the signal sent by P_i that results from this commitment.*

Proof: According to the algorithm, P_i checks that P_j is in the WAITING state before trying to commit to P_j. Let us assume P_j is in transaction

T_s when P_i sees P_j in the WAITING state. Therefore, it only remains to be shown that P_j is still in the WAITING state when P_i commits, as well as when the signal is received. This must be the case, however, because once P_j enters the WAITING state, it cannot change state until it first receives a signal. By lemma 2a, there were no signals pending when transaction T_s began. By lemma 3 no process other than P_i will commit to P_j during this transaction, so no signal other than P_i's are sent to, or received by P_j during this transaction. Therefore, P_j cannot unblock from the *WaitForSignal* operation and therefore cannot change state until receiving the signal sent by P_i. ∎

The preceding lemma shows that arbitrarily long delays may occur from the time P_i observes that P_j is in the WAITING state until P_i's signal actually arrives at P_j. If the commit succeeded, this lemma guarantees that nothing "interesting" will happen at P_j from the time P_i found it to be waiting until the signal was received.

Lemma 5 *No signals are lost in the alternative algorithm.*

Proof: By lemma 2a, no signals are pending at the beginning of each transaction. By lemma 3, at most one process can commit during a transaction, so at most one signal is sent (and therefore received) during a transaction. Thus, a signal can never arrive during a transaction while another has already been received but is still pending, so no signals are ever lost during a transaction.

No signals destined for a process P_j are lost between successive transactions of P_j because none can be sent to P_j while it is in the RUNNING state. This is true because (1) a signal is only sent to P_j following a commit operation (lemma 1), (2) Pj must have been in the WAITING

state when the commit occurred (lemma 4), and (3) P_j must remain in the WAITING state until the signal is received and absorbed by a *WaitForSignal* operation (lemma 4). ∎

Theorem 1 *If $P_i(T_r)$ signals (rendezvous) P_j, then P_j must be in some transaction T_s both when the signal is sent and when it is received. Further, $P_j(T_s)$ rendezvous $P_i(T_r)$.*

Proof: . By lemma 4, P_j must be in a transaction when the signal is sent and when it is received, and remain in the WAITING state during this period. By lemma 5, P_i's signal cannot be lost. By lemmas 1, 2a and 3, this is the only signal received by P_j during transaction T_s, eliminating the possibility of P_j accepting another signal instead of P_i's. Because P_j always executes *WaitForSignal* when in the WAITING state, the signal from P_i must be received, implying P_j rendezvous with P_i. ∎

Theorem 2 (Safety) *If $P_i(T_r)$ commits to $P_j(T_s)$, then the following properties must be true:*

1. *(Mutual consent) $P_i(T_r)$ rendezvous $P_j(T_s)$ and $P_j(T_s)$ rendezvous $P_i(T_r)$. In other words, the two communicating parties agree each is rendezvousing with the other.*
2. *$P_j \in GuardList_i(T_r)$ and $P_i \in GuardList_j(T_s)$.*
3. *Communications between $P_i(T_r)$ and $P_j(T_s)$ are compatible.*
4. *P_i and P_j will eventually communicate, complete their transaction, and return to the RUNNING state.*

5. *There does not exist a third process P_k ($k \neq i$ and $k \neq j$) such that $P_k(T_t)$ rendezvous with $P_i(T_r)$ or $P_k(T_t)$ rendezvous with $P_j(T_s)$.*

Proof:

1. $P_i(T_r)$ commits to $P_j(T_s)$, implying $P_i(T_r)$ signals $P_j(T_s)$ (lemma 1). This in turn implies the mutual rendezvous according to theorem 1.

2. The first part, showing $P_j \in GuardList_i(T_r)$, can be proved by contradiction. Suppose $P_j \notin GuardList_i(T_r)$. Then P_i would not have committed to P_j because P_i only scans those processes in $GuardList_i(T_r)$ (see the **FOR** loop in the *TryAlternative* procedure), contradicting our original assumption that P_i committed to P_j.

   It only remains to be proven that $P_i \in GuardList_j(T_s)$. It is seen from the algorithm that P_i checks $AltList_j$ just before committing to P_j, and $AltList_j$ is set to hold $GuardList_j(T_s)$ just before P_j enters the WAITING state, and therefore before the commit. However, an arbitrarily long delay may elapse from the time P_i checked $AltList_j$ to the time it committed. We therefore need to confirm that the value of $AltList_j$ that P_i checked is $GuardList_j(T_s)$ rather than $GuardList_j(prev^m(T_s))$ for some $m > 0$. This will be proven by contradiction.

   Suppose P_i checked $GuardList_j(prev(T_s))$. This would imply that the following sequence of events must have occurred:

   (a) $P_i(T_r)$ checks $GuardList_j(prev(T_s))$ (stored in $AltList_j$);

   (b) $P_j(T_s)$ modifies $AltList_j$ so that it becomes $GuardList_j(T_s)$;

   (c) $P_j(T_s)$ sets $WakeUp_j(T_s)$ to 0: and

(d) $P_i(T_r)$ commits to $P_j(T_s)$.

Event (a) must take place by the aforementioned assumption, and event (d) must take place by our original assumption that $P_i(T_r)$ commits $P_j(T_s)$. Event (c) must precede (d) because $WakeUp_j(T_s)$ must be reset to 0 before any commitment to $P_j(T_s)$ can occur (see definition of commit). Event (b) must precede (c) according to the order in which operations are performed in the algorithm. Event (b) must follow (a) in order to satisfy our supposition that P_i checked $GuardList_j(prev(T_s))$. However, this sequence of events is not possible because the locking protocol of the procedure *CheckAndCommit* (used by P_i when checking $AltList_j$) ensures that $AltList_j$ is not modified after P_i checks it (event (a) above), but before P_i commits (event (d)). Therefore, event (b) could not have occurred between (a) and (d), so our assumption that $P_i(T_r)$ examined $GuardList_j(prev(T_s))$ must be incorrect. Similarly, it is not possible that $P_i(T_r)$ examined $GuardList_j(prev^m(T_s))$ for any $m > 0$.

3. Compatibility is checked when $P_i(T_r)$ checks that it is in $AltList_j(T_s)$. Similarly, this information is implicitly updated whenever $AltList_j$ is updated. Therefore, this condition is satisfied using the same proof as was used in (2) to show P_i is in $GuardList_j(T_s)$.

4. Once rendezvous occurs between $P_i(T_r)$ and $P_j(T_s)$, each process initiates a communication with the other. Properties (2) and (3) above and the reliability assumption regarding the communication mechanism guarantee that the communication succeeds. Once this occurs, completion of the alternative operation immediately follows.

5. Suppose $P_k(T_t)$ rendezvoused with either $P_i(T_r)$ or $P_j(T_s)$. Recall a rendezvous occurs by either sending or receiving a signal to or from another process (definition of rendezvous), so there are four possibilities:

   (a) $P_k(T_t)$ received a signal from $P_i(T_r)$;

   (b) $P_k(T_t)$ received a signal from $P_j(T_s)$;

   (c) $P_k(T_t)$ sent a signal to $P_i(T_r)$; or

   (d) $P_k(T_t)$ sent a signal to $P_j(T_s)$.

   We need not consider signals sent before T_r, T_s, or T_t but received during these respective transactions because none can be pending when the transaction begins (lemma 2a).

   (a) Suppose $P_k(T_t)$ rendezvoused because it received a signal from P_i *during* T_r (signals generated by P_i outside T_r are not relevant). This implies $P_i(T_r)$ sent signals to *two* processes because our original assumption is that $P_i(T_r)$ committed to (and therefore signaled according to lemma 1) $P_j(T_s)$. It is clear from the algorithm that a process can signal at most one other process on any given transaction because any time a signal is generated, the transaction always completes without calling the *Signal* procedure again (see figure 2.4). Therefore, $P_k(T_t)$ could not have received a signal from $P_i(T_r)$.

   (b) Suppose $P_k(T_t)$ received a signal from P_j *during* T_s (signals generated by P_j outside T_s are not relevant). This implies $P_j(T_s)$ both sent a signal to P_k and received a signal from P_i within a single transaction. If $P_j(T_s)$ sent a signal, then, according to the algorithm in figure 2.4, P_j must have rendezvoused and completed

the transaction without ever entering the WAITING state or setting $WakeUp_j(T_s)$ to zero. This contradicts our original assumption that $P_i(T_r)$ committed to $P_j(T_s)$.

(c) Suppose $P_k(T_t)$ signaled $P_i(T_r)$. This implies $P_i(T_r)$ both sent a signal to P_j and received a signal from P_k within a single transaction. This latter signal must have been preceded by $P_k(T_t)$ committing to P_i (lemma 1). This commit must have occurred *during* or *before* T_r. But, $P_k(T_t)$ could not have committed to P_i during T_r because $WakeUp_i$ is never equal to zero during T_r. This is because, by assumption, $P_i(T_r)$ commits to $P_j(T_s)$, so $P_i(T_r)$ never enters the WAITING state (It is only then that the *WakeUp* variable is set to 0.) Also, $P_k(T_t)$ could not have committed to P_i before T_r and signaled P_i during T_r because this would violate lemma 4. Therefore $P_k(T_t)$ could not have sent a signal to $P_i(T_r)$.

(d) Finally, $P_k(T_t)$ could not have committed (and therefore could not have signaled) P_j *during* T_s because this would imply both P_k and P_i committed to P_j within a single transaction, violating lemma 3. $P_k(T_t)$ could not have committed to P_j before T_s and signaled P_j during T_s because this would again violate lemma 4. Thus, $P_k(T_t)$ could not have signaled $P_j(T_s)$ either. Therefore, $P_k(T_t)$ could not have rendezvoused with either $P_i(T_r)$ or $P_j(T_s)$, so the proof is complete. ∎

Note from the proof of (2) in the Safety theorem that it is crucial that accesses to *AltList* are controlled by locks, and that the act of checking the *AltList* and committing is atomic to ensure correct operation. Also note that the status of P_j may change immediately after P_i checks it. The algorithm operates correctly despite this inconsistency.

3.3 The Liveness Property

The liveness property guarantees that no deadlock or livelock situations can arise within the alternative algorithm. Such situations can only be caused by an erroneous *application* program. Lemmas 6 through 11 and theorem 3 prove that the liveness property is maintained by the proposed algorithm.

Lemma 6 *A process P_i will never return to the* RUNNING *state after entering a transaction unless a rendezvous occurred.*

> **Proof:** By inspection of the alternative algorithm, the process only returns to the RUNNING state when either: (a) $P_i(T_r)$ signals $P_j(T_s)$ or (b) after $P_i(T_r)$ receives a signal from $P_j(T_s)$. In either case, $P_i(T_r)$ rendezvoused with $P_j(T_s)$. ∎

Lemma 7 *A process P_i cannot remain blocked on a* Lock *operation in the alternative algorithm for an unbounded amount of time.*

> **Proof:** The only *Lock* operation performed by the algorithm is to serialize accesses to *AltList*. However, once any process obtains a lock on any *AltList*, it must eventually release that lock because no unbounded loop or blocking primitive is executed before the corresponding *Unlock* is performed. Therefore, the lock cannot remain in place for an unbounded amount of time. No process will remain blocked attempting to obtain a lock for an unbounded amount of time because every lock will eventually be unlocked, and the the *Lock* primitive is assumed to be fair. ∎

Lemma 8 *Suppose $P_i \in GuardList_j(T_s)$ and $P_j \in GuardList_i(T_r)$, and their respective I/O guards are compatible. P_i and P_j cannot both enter the* WAITING *state during transactions T_r and T_s, respectively.*

Proof: Proof by contradiction. Suppose both P_i and P_j enter the WAITING state on T_r and T_s, respectively. Because P_i reached the WAITING state, it must be the case that the *last* time P_i scanned the state of P_j before P_i entered the WAITING state, $State_j$ was either (1) RUNNING, (2) SLEEPING, or (3) WAITING but P_i failed to commit to P_j (If P_i successfully committed, they would have rendezvoused and completed the transaction according to theorem 2.) Consider the third case. We will now show that P_j must have been in a transaction *preceding* T_s for this case to apply. $WakeUp_j(T_s)$ is set to 0 *before* $State_j$ is set to WAITING. Therefore, if P_i saw P_j in the WAITING state while P_j was in transaction T_s, and P_i failed when it tried to commit, then it must be that some third process must have committed to P_j *during* T_s (after $WakeUp_j(T_s)$ is set to 0 but before P_i attempted to commit). But this successful commit must have resulted in a rendezvous, contradicting our original assumption that P_j blocked indefinitely in the WAITING state while in T_s. Therefore, if case (3) applies, P_j must have been in a transaction *previous* to T_s when P_i observed it to be in the WAITING state.

Similarly, P_j also reached the WAITING state, so P_i must have been in the RUNNING, SLEEPING, or WAITING state for a *previous* transaction the last time P_j scanned P_i before P_j entered the WAITING state. P_i and P_j could not have both scanned each other at the same instant because each would have found each other in the ALT state. Therefore, one scanned the other first. Without loss of generality, let us assume P_i scanned P_j first. $P_i(T_r)$ was in the ALT state when it scanned P_j, and because it did not rendezvous or abort (the latter would require P_j to be scanned again, making this *not* the last time P_i scanned P_j), P_i

must have remained in the ALT state until it changed to the WAITING state and blocked indefinitely. Therefore, when P_j later scanned P_i for the last time, P_j must have seen P_i in either the ALT or the WAITING state for transaction T_r. However, this contradicts the fact that P_j saw P_i in the RUNNING, SLEEPING, or WAITING state for a previous transaction. Therefore, the original hypothesis that P_i and P_j both entered the WAITING state must be false. ∎

Lemma 9 *A process P_i cannot remain continuously in the* ALT *state during a single transaction T_r for an unbounded amount of time.*

Proof: A process remains in the ALT state while it is scanning the processes in its *GuardList* trying to find one which is ready to rendezvous. If none is found, the process proceeds to the WAITING state. Because *GuardList* is necessarily bounded in length, we must show that a process does not spend an unlimited amount of time scanning a particular guard.

P_i moves on to the next *GuardList* entry or eventually changes state when it finds the process corresponding to the current guard is in either the SLEEPING, RUNNING, or WAITING state. Therefore, we only need to consider scanning a process P_j which is also in the ALT state. If $TransID_j < TransID_i$, then P_i aborts *TryAlternative* and changes to the SLEEPING state. Thus we need only examine the case $TransID_i < TransID_j$ (both cannot have the same ID). In this case, P_i enters a loop waiting for $State_j$ to change. In order for P_i to remain in this loop an unbounded amount of time, P_i must continually sample P_j while $State_j$ is ALT. There are three ways P_i's samples can indicate P_j remains in the ALT state for an unbounded amount of time: (1) P_j is

also locked into the ALT state for an unbounded amount of time; (2) P_j repeatedly aborts *TryAlternative*, changes to the SLEEPING state, and then retries *TryAlternative* (changing back to the ALT state) in perfect synchrony with P_i's samples of $State_j$; or (3) P_j repeatedly rendezvous, changes to the RUNNING state, and then initiates a new alternative operation in perfect synchrony with P_i's samples of $State_j$. These are exhaustive because a process can only return from *TryAlternative* after a rendezvous or after an aborted attempt. Case (2) cannot occur, however, because the sleep period is set to a time sufficiently large that successive samples by P_i will detect that P_j is in the SLEEPING state. Similarly, case (3) cannot occur because the minimum execution time of the *Send* and *Recv* primitives are assumed to be larger than the time between successive samples of the polling loop. Therefore, only case (1) remains.

The previous discussion shows that P_i can only remain in the ALT state scanning P_j an unbounded amount of time if the following conditions hold: (1) $TransID_i < TransID_j$, and (2) P_j remains continuously in the ALT state on the same transaction an unbounded amount of time. By the same argument presented above, P_j will only remain in the ALT state on a single transaction an unbounded amount of time if some other process P_k is in P_j's *GuardList*, $TransID_j < TransID_k$, and P_k remains continuously in the ALT state an unbounded amount of time. Continuing this logic, because the number of processes is bounded, the original process P_i will only remain in the ALT state for an unbounded time if a *cycle* of processes exists such that each is waiting for the next process in the cycle to leave the ALT state. This would require that $TransID_i < TransID_j < TransID_k < \cdots < TransID_i$, which is

clearly not possible. Therefore, no such cycle can exist, so P_i cannot remain continually in the ALT state for an unbounded amount of time.

∎

Lemma 10 *The* TryAlternative *procedure cannot return* FAILED *an unbounded number of times during a single transaction T_r in some process P_i.*

Proof: *TryAlternative* returns FAILED if and only if P_i scans another process P_j and finds P_j is also in the ALT state, and $TransID_j < TransID_i$. The number of guards in *GuardList* is finite, so if procedure *TryAlternative* fails an unbounded number of times, it must be that for some process P_j, the conditions $State_j =$ ALT and $TransID_j < TransID_i$ persist for an unbounded amount of time.

P_j cannot remain continually in the ALT state for an unbounded amount of time in a single transaction (lemma 9). Therefore, it must be the case that either (1) P_i finds P_j in the ALT state for a *different* transaction an unbounded number of times; or (2) within a single transaction, P_j repeatedly switches back and forth between the ALT and SLEEPING states for an unbounded number of times, and it so happens that every time P_i retries *TryAlternative* and scans P_j, P_i finds that P_j is in the ALT state. In case (2), *TryAlternative* must fail an unbounded number of times in P_j as well as P_i.

Case (1): This is not possible because each new transaction ID is larger than all previous IDs. If P_i finds P_j in the ALT state for a new transaction an unbounded number of times, this would imply there are an unbounded number of transaction IDs less than $TransID_i$. This cannot be the case because transaction IDs are positive integers.

Case (2): An argument similar to that used in lemma 9 can be used here. Summarizing the arguments presented thus far in this lemma, *TryAlternative* in P_i will only fail an unbounded number of times if it also fails an unbounded number of times in some other process P_j, where $TransID_j < TransID_i$. Similarly, P_j will only continue to fail if some other process P_k exists which also continues to fail, and $TransID_k < TransID_j$. Because the number of processes is bounded, a cycle of processes must exist such that $TransID_i > TransID_j > TransID_k > \cdots > TransID_i$, which of course, cannot occur. Therefore, a process cannot fail the *TryAlternative* procedure an unbounded number of times. ∎

Lemma 11 *For each alternative operation initiated by P_i, P_i eventually either rendezvous with some other process P_j and returns to the* RUNNING *state, or moves to the* WAITING *state. In other words, a process cannot remain in the* ALT *state in the same transaction for an unbounded amount of time.*

Proof: The only way a process can *not* reach the WAITING state or rendezvous is to remain continually in the ALT state, or switch back and forth between ALT and SLEEPING an unbounded number of times. The latter case implies *TryAlternative* fails an unbounded number of times within a single transaction. Neither is possible according to lemmas 9 and 10. ∎

Theorem 3 (Liveness) *Suppose two processes P_i and P_j each initiate an alternative operation and $P_j \in GuardList_i(T_r)$ and $P_i \in GuardList_j(T_s)$ and their communication requests are compatible. If neither P_i nor P_j rendezvous with another process during their respective transactions, P_i and P_j will eventually rendezvous with each other during T_r and T_s, respectively.*

Proof: According to lemma 11, P_i and P_j must each eventually either rendezvous or enter the WAITING state. They both cannot enter the WAITING state according to lemma 8. Therefore, at least one of the two processes, say P_i, must rendezvous. By assumption, P_i cannot rendezvous with any process other than P_j, so P_i must rendezvous with P_j. By theorem 2, P_j must also rendezvous with P_i. Therefore, P_i and P_j must eventually rendezvous with each other. ∎

3.4 The Termination Protocol

As mentioned in section 2.4.5, a terminating process must perform some "clean-up" work before it exits. It is vital that the termination protocol (1) not interfere with non-terminating tasks, and (2) guarantee that all repetitive contructs that should terminate eventually do. The first criterion is analagous to the safety requirement and the second to the liveness requirement.

The termination protocol employs a special signal distinct from the rendezvous signals sent between the processes. The first requirement above is met based on the following intuition. A special signal is sent by a terminating process to a process in the WAITING state *only* when the terminating process has decided that no other processes will ever send, or have sent, the WAITING process a similar special signal. Therefore, the special signal will always be received and absorbed.

The lemma below validates that a terminating process has made the correctly sends special signals.

Lemma 12 *If a terminating process P_j finds, in its cleanup phase described in section 2.4.5, that $GuardCount_i$ of a* WAITING *process P_i in transaction T_r equals 0 after P_j has atomically decrmented it by 1, then none of the possible communicants of $P_i(T_r)$ other than P_j will send a signal to $P_i(T_r)$.*

Proof: There are two ways that $GuardCount_i(T_r)$ can be decre-

mented. (1) By $P_i(T_r)$ itself, when it finds some communicants in the TERMINATED state during transaction T_r. (2) By some terminating communicant of $P_i(T_r)$ in cleanup phase. $GuardCount_i(T_r)$ can notbe decremented twice due to the termination of a single communicant. When $GuardCount_i(T_r)$ is decremented by P_i itself, it excludes the communicant in $AltList_i(T_r)$. Because the decrement is done with $State_i(T_r)$ remaining in the ALT state, it effectively prohibits the communicant from decrementing $GuardCount_i(T_r)$. On the other hand, if the communicant successfully finds itself in $AltList_i(T_r)$ and decrement $GuardCount_i(T_r)$, then $P_i(T_r)$ must have seen the communicant in a state other than TERMINATED and could not have decremented $GuardCount_i(T_r)$ itself.

According to the above argument, the communicant P_j, under the predicated condition, must be the only possible communicant to send the special signal to $P_i(T_r)$.

An argument similar to that in the Lemma can be developed to guarantee that, if $P_i(T_r)$ finds $GuardCount_i(T_r)$ has been decremented down to 0 before it enters the WAITING state, then it must have excluded all the communicants in $AltList_i(T_r)$, effectively prohibiting all its communicants from sending signals. It, then, correctly exits the repetitive construct without leaving any signals pending in the last transaction.

The proof of satisfaction of the second requirement described earlier is straightforward. Intuitively, if a process could not determine itself when it should exit the repetitive construct during the "last" transaction, then one and exactly one of its communicants of this last transaction will make this determination. This point can be clearly asserted given the arguments above.

Chapter 4

A Butterfly Implementation

4.1 Overview

The implementation of A CSP system, based on the algorithm described earlier, has been carried out on an 18 nodes BBN Butterfly Parallel Processor™. A CSP program consists of several C procedures, each representing a CSP process. A distinct C procedure is identified as the "root" process and the entry point of the user program. This procedure typically spawns the other processes. Each process can invoke the parallel, alternative or repetitive operations using a C syntax that will be described later. The features mentioned in chapter 2 such as I/O guards and automatic termination of repetitive commands are provided As will be seen later, processes can be created *dynamically*, as contrast to Hoare's original definition in which processes are *static*. This property is due to the fact that processes communicate using explicit channels, which can be created dynamically.

All application programs are created, complied and linked on a front end machine, usually a VAX, and then downloaded to the Butterfly for execution. The user need only learn the C language interface described in section 4.3. Some examples are shown in Appendix A and B. The Butterfly allows a user to declare exclusive access to a group of processors nodes, called the *user cluster*, in a session. The CSP system distributes the "processes" to any subset of the user cluster that is specified in each run.

4.2 The BBN Butterfly Parallel Processor™

The BBN Butterfly Parallel Processor™ is a MIMD, shared memory multiprocessor that uses an Omega network to interconnect its *processor nodes*. Each processor node contains a micro-coded controller called the PNC which controls all the memory references from either the local processor (a Motorola 68020) or from the Butterfly switch(the Omega network). The PNC also facilitates atomic operations. Typical operations include atomic arithmetic manipulations on a given memory address, event posting/receiving, and queue operations. All of these operations are defined relative to *processes* instead of *processors*, so intra-processor and inter-processor interactions appear the same.

Remote memory references in the Butterfly are 6 to 7 times longer than that of a local access under light system load. In contrast,message-based architectures such as the hypercube requires many milliseconds for interprocessor communication.

A Butterfly may contain up to 256 processor nodes, each with up to 4 Mega bytes of memory. The current Utah configuration contains 17 nodes, 15 with 1 Mega byte of memory, the other 2 with 4 mega bytes. The shared memory scheme is achieved through the used of *Segment Attribute Registers*(SARs) which map virtual addresses into physical memory locations. Each processor node is equipped with 512 such SARs and each SAR is capable of addressing up to 64 Kilo bytes of memory. The 24 bits virtual address now being used restricts any *process* to having at most 256 segments(256 SARs) mapped in its address space at one time, allowing the process to address 16M bytes of memory without dynamically swapping SARs. The restriction is due to the need of compatability to the older 68000 based processor nodes, which uses 24 bit virtual addresses. Newer versions of the Butterfly will use the 32 bit address bus of the 68020, providing a 4 Giga bytes address space.

The programming language C is used for this implementation. Butterfly C is identical to Unix C with a few extensions. It has an enumerate data type similar

to that in Pascal. It also provides an exception handling mechanism similar to the Lisp catch/throw construct. This construct allows nonlocal go-to that proved to be useful in the CSP implementation. The catch/throw construct is defined as a macro and expanded at compile time by the language pre-processor.

Currently, the Butterfly serves as a back-end processor. A 10 MB Ethernet link connects it to the host (VAX 8600) Therefore, when a command is issued to run a user program, Chrysalis searches a specific directory on the host machine for the program 'template'(executable image) and loads it across the network for execution on the Butterfly. All program development, compilation and linkage are done on the host machine. Other technical information can be found in [2,4,6,25,27]

4.3 Language interface

The programming language C was chosen, since it is the language in which most of the operating system is written. C is also well know and widely used.

The environment needed for correct CSP operations is set up automatically before the user program is entered, i.e., the user does not need to invoke any system initialization routine. Although this is desirable, it does necessitate a smaller discrepancy from ordinary C programming style. The "main" procedure of the application program must be a C procedure named "cspUser" linked with the system object files.

The various CSP constructs are available to the application programs through a collection of C procedure and macro calls. A system header file "cspUsr.h" must be included at the beginning of each user source file. Figure 4.1 summarizes the syntax of these calls.

4.3.1 The Parallel Construct

The "CSP_PARALLEL_BEGIN(n)" and "CSP_PARALLEL_END" macros are used in conjunction with the procedure CreateP() to create CSP processes.

```
    /* alloc port ids for channel */
PROC VOID  AllocChannel(&chnl)
CHANNEL  chnl;

    /* a 'co-begin' 'co-end' pair */
MACRO VOID  CSP_PARALLEL_BEGIN (n)
int n;

MACRO VOID  CSP_PARALLEL_END

    /* create a process within PARALLEL */
PROC VOID CreateP(fPtr, fmtStr, param, ...)
PROC VOID *fPtr();
char *fmtStr;
int  param;

    /* The alternative entry call */
PROC int Alternative(argblk)
int **argblk;

    /* The repetitive construct */
MACRO VOID CSP_REPEAT_BEGIN

MACRO VOID CSP_REPEAT_END
```

Figure 4.1: Interface of The CSP Procedure Calls

parent process, i.e., the process that calls the CSP_PARALLEL constructs, will suspend execution until all of its children have completed and terminated. The parameter n to the macro CSP_PARALLEL_BEGIN is the number of processes to be created in this call, and should match the number of times that the procedure CreateP() is called between the BEGIN and END statements.

Each call to the procedure CreateP() causes the creation of a process that executes (at least potentially) in parallel with others created in the same parallel construct. The first parameter is a function pointer which points to a C procedure constituting the process body. The second parameter is a "format" string with each character indicating the type of a parameter passed to the process. This string also *implicitly* specifies the total number of parameters that are passed. The third and subsequent parameters are the values passed to the process, in the order specified by the format string. Any C looping constructs such as FOR and WHILE can be used to repeatedly generate calls to CreateP(), providing convenient ways of instantiating a process many times with different parameters. The processes communicate with each other through *unilateral channels*. Though technically different from Hoare's[18] direct naming scheme, these two communication models are logically equivalent. The channel model provides more flexibility in the hierarchical composition of smaller systems into larger ones. Figure 4.2 shows an example of how the construct is used.

A channel is a point to point communication link that connects exactly two processes. Each end point is called a *port*. A channel is unilateral in that one of its ports can only receive messages while the other can only transmit. Each port is represented by a unique number and each channel is represented by a pair of ports. In C, the channels are defined as a "CHANNEL" structure whose components are two "PORT"s, named "in" and "out". A process can receive ports as parameters as well as arrays of ports. The latter is a convenient way of passing a variable

```
cspUser()     /* The root process.. */
{
  CHANNEL  ch;
  ..........

  AllocChannel(&ch);
  CSP_PARALLEL_BEGIN (2)
     CreateP(proc1, "p", ch.in);
     CreateP(proc2, "p", ch.out);
  CSP_PARALLEL_END
  ..........
}

PROC VOID proc1(inport) { PORT inport; .... }

PROC VOID proc2(outport) { PORT outport; .... }
```

Figure 4.2: An Example of Calling the Parallel Command

number of ports to a process. A port is specified in the format string of createP() by a character 'p', while a NULL terminated array of ports is specified by an 'a'. Other parameters that are not concerned with the channel connections are denoted by an 'o'. Channels are allocated using the procedure AllocChannel(), whose only parameter is a pointer to a CHANNEL structure. Upon return from a call, the "in" and "out" component of the channel is ready to be passed to the children. In this paradigm, all ports used in a process for communication are allocated by its parent and passed to the process as parameters when it is created.

4.3.2 The Alternative Construct

To invoke an alternative operation, a process calls the procedure Alternative() and passes to it a pointer to an argument block that specifies the ports involved in the operation, along with their respective modes, message size and buffer address. Figure 4.3 shows the structure of the argument block.

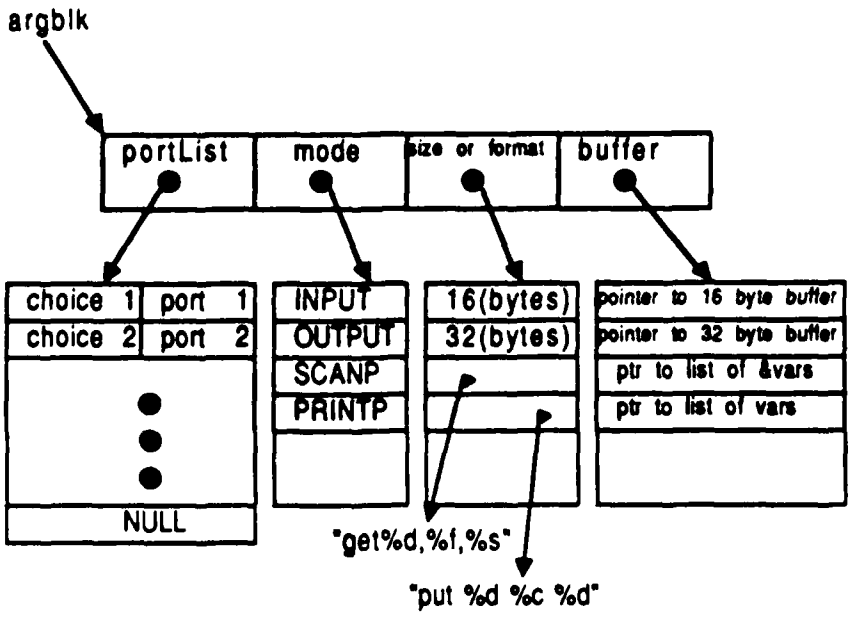

Figure 4.3: The Argument Passed to Alternative

The modes associated with each port can be "CSP_INPUT", "CSP_SCANP", "CSP_OUTPUT", and "CSP_PRINTP". The first two modes are only allowed on input ports, and the last two only on output ports. The INPUT and OUTPUT modes are block transfer modes and the size of the communication is in bytes. The SCANP and PRINTP modes are "formatted" transfer modes similar to the C scanf and printf functions. In this case, the size of the communication is implied in the format string, and the buffer is a list of variables(pointers in the SCANP case), as in the C scanf and printf functions. The mode CSP_INPUT is *compatible* only to CSP_OUTPUT and CSP_SCANP only to CSP_PRINTP. An error will result if two processes connected by a channel use incompatible modes on the ports in an alternative operation. A simple communication command that does not involve any alternatives is in fact a special case of the alternative operation. For the sake of consistency and integrity, it must be done using an alternative call with only one "alternative".

The ports involved in an alternative operation are passed to the procedure Alternative() in the form of a list. Each port in the list is *tagged* with a non-negative number which will be returned by the procedure if the corresponding port has been selected by the alternative operation to perform the communication. This number can be used in a case statement to dispatch control to the appropriate code. However, if the tag associated with a port is negative, the port is *disabled* and will not be considered for selection in this alternative operation. The procedure Alternative() will raise an exception when all of the ports passed to it are disabled. This exception will be interpreted by the underlying system as an error signal unless the application program turns it off by means of the *repetitive* construct described in the next section. Note that a port is also disabled when the process that owns the other port of the channel has terminated. This is the automatic termination semantics of CSP[18] and is useful in some applications(see examples in the Appendix).

In the current implementation the CSP_PRINTP and CSP_SCANP operations are not available, but they can be easily added.

4.3.3 The Repetitive Construct

The "CSP_REPEAT_BEGIN" and "CSP_REPEAT_END" are macros used to capture and handle the exception raised by the Alternative() procedure. When this construct surrounds an alternative construct, it specifies the indefinite repetition of the alternative operation. The repetition is completed when all of the ports passed to it are disabled either explicitly by the process itself or implicitly because the communicants have terminated. Control is then transferred to the statement immediately following the CSP_REPEAT_END. It is important to note that errors due to incompatible I/O attempts, as described in the previous section, cannot be caught by the repetitive construct because they indicate an error condition.

```
SomeProcess(..ports..)
{
  PORT  ..ports..;
  int   choice;
  ..........
  ... set up ArgumentBlk ...

  CSP_REPEAT_BEGIN
     choice = Alternative(ArgumentBlk);

     switch (choice) {
     case  CASE_1 : {.... Action 1 ....}
     case  CASE_2 : {.... Action 2 ....}
     .........

     }  /* end switch statement */
  CSP_PARALLEL_END
  ..........
}
```

Figure 4.4: An Example of Calling the Repetitive Command

An application program cannot specify the number of times a repetitive construct should make the alternative operation execute. However, the program can disable all of the ports when it wants to terminate the repetitive construct. Figure 4.4 shows an example of its usage.

4.4 Implementation Issues

4.4.1 Some Considerations

The most fundamental entities in CSP are processes. On the Butterfly, Chrysalis provides a mechanism for creating processes from a user program, but it has several drawbacks.

First, Chrysalis processes are best suited for large-grain computations, whereas CSP processes can be quite small. The overhead of creating a Chrysalis process for each CSP process is excessive. For example, in Appendix B, the producer process

in the bounded-buffer example repeatedly generates a datum and sends it to the buffer process in a loop. Defining each such simple operations as a C procedure and create a Chrysalis process for each one will be very inefficient.

Secondly, there may exist a large amount of communication between the CSP processes. However, each Chrysalis process has it own virtual address space The Map_Obj and Unmap_Obj system calls provides a way of attaching memory segments that are globally sharable to a process's address space. However this scheme is quite costly and is not feasible on a per-communication basis. Furthermore, since there are only a limited number of SARs on each processor node, and each Chrysalis process requires a certain minimum number of SARs even if it does not do any memory object Mapping/Unmapping, only a few processes can co-exist on a processor node at a time.

An efficient means of creating light-weight processes is required in which each process, upon creation, shares part of its virtual address space with common, global area. *Coroutines*[12,37] seem a suitable solution for implementing *CSP processes*. Coroutines are multiple threads of control within one Chrysalis process that relinquish execution *voluntarily* to others by calling a procedure that saves, restores and switches between contexts of the threads. This scheme requires a stack for each thread, and the ability to create and transfer of control from one thread to another. Here coroutines will be called *tasks* to avoid confusion with the Chrysalis processes.

This implementation hides peculiarities such as the coroutine mechanism from the application as much as possible. Otherwise, the application programmers would be required to call certain "system" procedures at the right time, in order to set up environments or to do necessary cleanup work.

4.4.2 System Startup

When the system is initiated, a Chrysalis process is loaded and run on one processor node selected by the user. This node is called the *master* node. The Chrysalis process running on the master node is called the master process. The master process determines the set of processor nodes(the *user cluster*) available to the application program, and creates a Chrysalis process on each of the nodes in the cluster, except the master node itself. Those processes are instances of the master template and will be called the *slave* processes. Each slave process is only slightly different code the master process. All processes will behave the same after a suitable environment has been established. Since globally sharable memory is distributed over all nodes, each process needs to allocate enough memory space on its node to accommodate

- a stack area for local coroutines(stack pool),
- a heap area for temporary buffers(local heap), and
- a heap area for globally shared memory (global heap).

The combined global heaps of all of the nodes forms a region that is accessible to all the processes, after they have all mapped the region into their individual virtual address spaces in exactly the same way. This region is called the *globally shared region*. The result of this mapping is that the virtual address space of each process is constituted of several logical segments:

1. The code segment, residing physically on the local node.
2. The data segment, residing physically on the local node.
3. The coroutine stack pool, residing physically on the local node.
4. The local heap, residing physically on the local node.

5. The globally shared region, with components scattered among the nodes of the user cluster.

Since all the processes map the shared memory region into exactly the same virtual location in their own address spaces, an address that is within the shared memory region in one process's space can be passed to another process and used there to access the same physical location. Figure 4.5 shows the addressing space as seen by the processes.

4.4.3 The Cooperating Schedulers

Once initialization is completed, each process is ready to schedule coroutines to run on the local node in much the same way a simple operating system schedules its user processes. The master scheduler then creates a task for the "main" user procedure on the local node, which usually spawns children task using the CSP parallel command (Refer to the Appendix for examples). In this implementation, the children are sent to other nodes for creation in a round-robin fashion. For example, if the N nodes in the user cluster are numbered from 0 to N-1, then the children are sent to 0, 1, 2, an so on. Each task is capable of spawning tasks using the parallel construct, and the round-robin dispatching is handled by the local scheduler. Upon creation of each task, the scheduler allocates a stack from the stack pool area and associates it with the task. It also allocates a block to hold various control information such as the task state, the alternative list, the *GuardCount* variable, etc. Since this block must be accessible by any other tasks, it is allocated from the globally shared region. The address of this block in global space can therefore be used as the "ID" that uniquely identifies the task associated with it. Figure 4.6 shows the contents of the task control block.

In the alternative algorithm presented earlier, tasks need to synchronize each other through a signal/wait mechanism. The synchronization between tasks on the

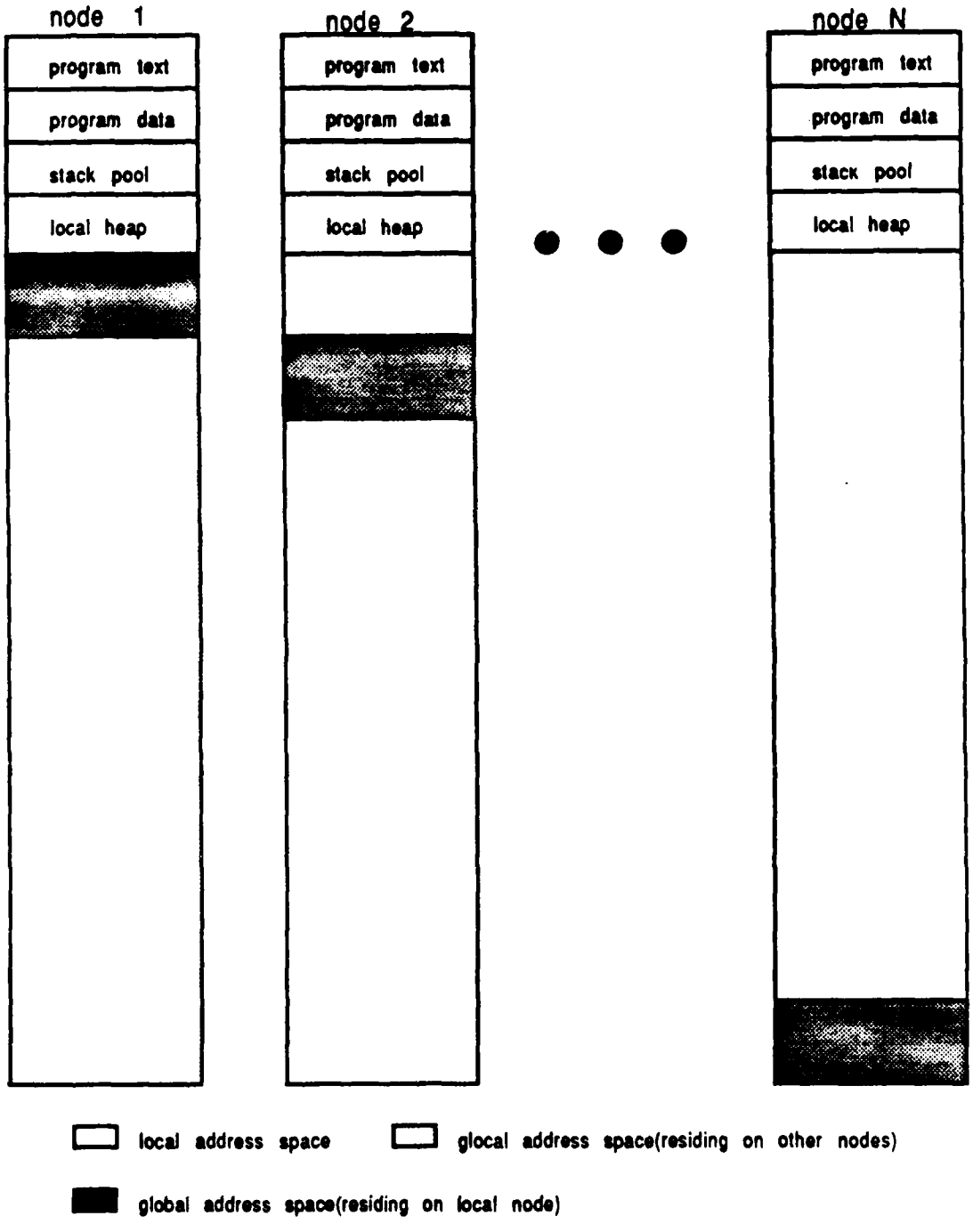

Figure 4.5: Address Space of The Processes

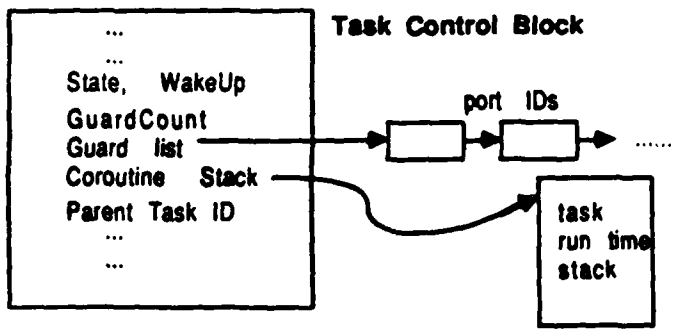

Figure 4.6: Structure of The Task Control Block

same node can be handled by the local scheduler alone, whereas synchronization between two tasks on different nodes requires the cooperation of the two schedulers involved. Therefore, each scheduler(process) has a queue associated with it to buffer synchronization signals from other schedulers. The queues are implemented using Chrysalis dual queues, whose operations are guaranteed to be atomic in this multiprocessor environment. Currently there are two kinds of interactions between schedulers: (1) Task creation requests, sent when new tasks are spawned and dispatched. (2) Signal requests, sent when tasks attempt to wake up tasks on other nodes. In the alternative algorithm the Lock/Unlock mechanism is used by the tasks to gain exclusive access to certain data structures. The time each task spends in the critical section is sufficiently short that a context switch is not desirable. Therefore, a simple busy wait loop that utilizes the Butterfly "test and set" primitive to determine the exit condition is employed. No queue operation is involved since it does not concern local scheduling and interactions between schedulers. Also the 'Sleep' mechanism can be handled by the local scheduler, so it does not concern cooperation with the others. However, whether a context switch should be performed when a sleep operation is executed depends on the length of the sleep period, and a decision can be made accordingly. For example, if the specified period is far greater than the coroutine transfer time, which is about 200 micro seconds

```
while(TRUE) {
   While(message queue not empty)
      switch (message type) {

      case MAKE_TASK :
        Allocate stack from local stack pool;
        Initialize stack;  /* Push parameters, etc.. */
        Allocate task control block from global shared region;
        Initialize task control block;
        Register task to the port table;
        Put task in ready list;

      case SIGNAL_TASK :
        find waiting task and put it in ready list;
      }  /* end of switch */

   Wake up expired sleeping tasks(put back in ready list)
   Transfer control to first ready task;
   /* control will be back when task waits or sleeps... */

}    /* End while TRUE  */
```

Figure 4.7: The Algorithm Executed by A Scheduler

in this implementation, the task can actually be de-scheduled. Otherwise, a busy wait loop suffices. A later chapter on fine tuning the performance will discuss how the sleep period is set for a wide range of workload conditions.

Figure 4.7 describes the algorithm executed by a scheduler. It polls its queue and processes all pending signals from the other schedulers , puts awaken tasks back into a list of 'ready' tasks, and transfers control to the first task in the ready list. The task that has regained control will eventually execute the alternative algorithm to communicate with other tasks and therefore may have to wait for a signal from other tasks(either on the same or a different node), at which time it relinquishes control back to the scheduler. The scheduler then polls its queue again and repeats the above steps.

When the "root" task terminates, the entire program also terminates. The master scheduler, which executes the root task, is the parent Chrysalis process of all other schedulers. When a Chrysalis process exits, the operating system will automatically kills all of its children process(perhaps running on other nodes) and releases all resources by the killed processes. Therefore, the identity of the master scheduler is retained from system startup, though it is not meaningful during the computation.

4.4.4 The Global Port Table

As described in previous sections, the *channel* and *port* model is used by the application program as the communication mechanism. The alternative algorithm must be able to locate a "remote" task on the other side of a channel, given the local port ID. The state and control information of the remote task can then be inspected and a decision can be made as to whether or not a rendezvous is possible.

The problem of locating tasks through port connections is greatly simplified on the Butterfly through the use of shared memory. It is sufficient to allocate a piece of global shared memory which can reside on an arbitrarily selected node, that records all the port-task associations. Figure 4.8 depicts such a port table in globally shared space. Port IDs are short integers where input and output ports are distinguished as taking on odd or even values. The reasons for this distinction are (1) it is easy to detect illegal operations on ports, e.g., output operations on input ports. (2) adjacent integers can be specified as the two ports of a channel, simplifying easy the location of one given the other. A channel allocation routine is available to application programs which need to allocate channels and pass the ports to their children. Upon request, this routine simply atomically increments a usage counter associated with the port table to allocate 2 adjacent entries.

It should be noted that, before a group of tasks can communicate with each

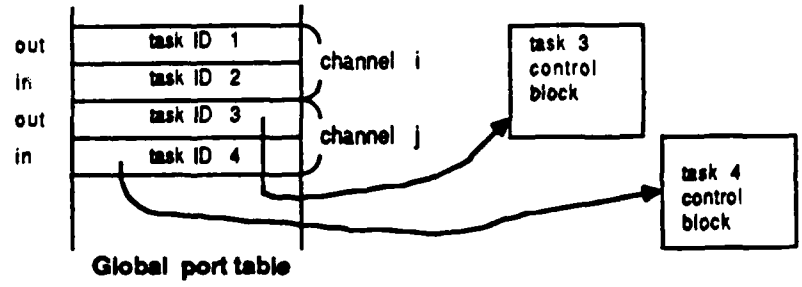

Figure 4.8: Structure of The Global Port Table

other through the alternative operation, each of them must make sure that all its possible communicants have completed putting their task IDs into the entries of the port table. This is very important because the result of relying on an inconsistent port table in an alternative operation can be disastrous. In this implementation the problem is resolved by having each task take responsibility for consistencies among it children. Every time a task spawns children tasks using the parallel construct. it waits for a signal returned by each of the children indicating all the them have *finished registration with the port table.* When this condition is true, it sends a "go-ahead" signal to them. This protocol is built into the schedulers and is transparent to application programs.

4.4.5 Automatic Termination

The termination protocol mentioned above has been incorporated in the alternative algorithm. There are two ways that a task in the alternative operation can determined when the termination condition is met, and the surrounding repetitive construct should exit. To emulate the behavior of a repetitive construct after the termination condition is satisfied, the alternative algorithm could return to the caller with a special value indicating such a situation and let the application program determine what to do afterwards. However, a better scheme would be to

utilize a *nonlocal go-to* mechanism like the LISP catch/throw. This scheme works as follows. Whenever a repetitive construct is encountered in the application program, a "catch" is executed. The alternative construct "throws" when the termination condition is satisfied, which is caught by the enclosing repetitive construct that has executed a "catch" earlier. The advantage of this scheme is that if a catch had not been executed earlier, then a throw by the alternative operation would cause an error according to the catch/throw semantics. In CSP, this indeed is an erroneous situation.

The Butterfly C language provides such a catch/throw mechanism. When a catch statement is executed in a procedure, a marker is placed into the current stack frame in the run time stack so that when the procedure returns the marker can be popped off the stack along with the rest of the procedure's frame. The marker can be thought of as a "continuation" where control is to go when a throw is caught by the catch. As the procedure makes further procedure calls that again execute catch statements, more markers are placed into the stack, and all these "outstanding" markers form a linked list according to the time they are created and placed. A pointer to the most recent outstanding marker is kept in a well known location, called the "catch pointer", so that whenever there is a throw by some procedure deep in the stack, the most recent marker can be found and used as the continuation. The catch pointer is then updated as the second most recent marker in the link. In the Butterfly environment, each Chrysalis process has exactly one catch pointer. Therefore, it is necessary to save the catch pointer as part of the status when context switching is performed between coroutines, since each coroutine represents a CSP process and may execute its own catch statements.

As manifested in the beginning of this chapter, the coroutine approach facilitates automatic system control that hides application programs from implementation details. Figure 4.9 summarizes the stages a coroutine experiences during its lifetime.

```
Phase 1:  Created by the scheduler.
          Registered to global port table.
Phase 2:  Notify parent registration done..
          Wait for ''go-ahead'' signal from parent.
Phase 3:  Running, or
          de-scheduled waiting for synchronization, or
          sleeping.
Phase 4:  Finished computation.
          Execute termination protocol.
Phase 5:  Inform parent of demise.
          De-scheduled permanently.

NOTE: Only phase 3 is specified by application
      program.  All others are done by the system
      upon creation or termination transparently.
```

Figure 4.9: The Phases of A Coroutine's Lifetime

4.4.6 Fairness

One issue regarding the alternative construct which has received considerable attention is *fairness*. In particular, two types of fairness, *weak* and *strong* fairness, have been defined [14,38]. We call an implementation of the alternative construct *weakly fair* if it can be guaranteed that during the infinitely repetitive execution of an alternative command, a guard that remains continuously available (i.e., enabled and the neighboring process is ready to communicate) will eventually rendezvous. An implementation is said to be strongly fair if the implementation guarantees that any guard which is available *infinitely often* (though not necessarily *continuously* as is the case in weak fairness) will eventually rendezvous.

The algorithm shown in figures 2.3 and 2.4 is not fair in either the weak or strong sense. However, weak fairness can be achieved by modifying the algorithm so that the order in which the *TryAlternative* procedure scans guards, which im-

plies a certain prioritization of the guards, varies from one call to the next so that each guard is eventually scanned first. More precisely, the alternative algorithm is modified so that it assumes there is a distinct integer variable associated with each alternative construct in a given CSP program, e.g., with the *mth* alternative construct in process P_i there is the variable $Alt_{i,m}$. Initially set to 0, this variable is incremented each time this particular alternative construct is executed. It therefore indicates the number of times P_i has invoked the corresponding alternative construct.

Also, the FOR loop in the *TryAlternative* procedure is modified to begin scanning guard $(Alt_{i,m} \bmod n)+1$ rather than the first guard, where n is the number of guards in the alternative construct. Note the FOR loop skips disabled guards, as indicated in the argument block shown in Figure 4.3. It executes up to n iterations, and the index variable of the FOR loop "wraps around" to 1 after scanning the nth guard. The number n can be included in the argument block so that algorithm won't have to calculate it every time.

These variables could be defined by the compiler or pre-processor. However, in this implementation the application program must define them. The example programs in the appendices show these "fair seed" variables. Note the algorithm has access to a globally defined variable called $\_fairSeed\_$, which should be set to the fair seed Alt_i before entering a specific alternative operation i. The application programs are also responsible for incrementing the respective fair seed after each repetition of the alternative operations.

It is not difficult to see that the algorithm is weakly fair. For instance, a process waiting to be served by another process will be granted the service within a finite number of times that the possible servers reach rendezvous. It is not strongly fair, however.

```
Communicate(theOtherTask, mode, size, data)
{
  if (size is 0)  return; /* do nothing */
  else
     if (mode is input) {
         Wait for data to be available;;
         Copy data;
         Acknowledge to theOtherTask;
     }
     else {      /* mode must be output */
         Send data;
         Indicate data available;
         Wait for acknowledge;
     }
}
```

Figure 4.10: The *Communicate* routine

4.4.7 Communication

When two tasks have reached rendezvous in their respective alternative operations, i.e., one sends a signal to the other, they exchange a message (or nothing but the synchronization effect) and exit the alternative operation. The routine *Communicate* shown in figure 2.4 does this for the local task, given the ID of the other task with which it is communicating, and other information such as I/O mode, data and size. Figure 4.10 outlines the *Communicate* routine.

The CSP model requires that all communication be synchronous, i.e., the sending party must not resume its computation until it is confirmed that the receiving party has received and digested the message. There are two cases to be considered. First, the size of communication is zero, i.e., the two parties are interested only in synchronization, but not in data transfer. Second, there is data transferred in addition to synchronization.

The first case is very simple, since the task originally in WAITING state is waken up by exactly one task which later communicates with it (refer to the proof

in chapter 3.) Therefore, the synchronization effect has been achieved when the awakening signal is sent to the WAITING task. The *Communicate* routine needs do nothing. The *Dining Philosophers* shown in Appendix B is an example of this kind of communication.

The second case is more complicated because the receiving party has to wait for the data to become available, and the sending party has to wait for an acknowledge from the receiving party once the data has been sent. These operations may cause context switching between tasks.

It should be noted that private data of user tasks can reside in the same logical address but physically distinct, if they are on different processor nodes. The only way to transfer private data is to use buffers in the shared memory between Chrysalis processes. The actions "Send data" and "Copy data" in figure 4.10 therefore indicate copying to and from the shared memory buffer, respectively.

4.5 Discussion

This section describes possible alternatives to the current implementation. However, they are not included in the current implementation.

4.5.1 Dynamic Load Balancing

The current implementation is based on a strictly static task allocation policy, i.e., each task (coroutine) stays on the same processor node and runs to its completion. One may argue a worker/task model is better for the sake of dynamic load balancing. This model, as employed by the Uniform System[35], uses only one global task queue among the workers(processors), and each worker obtains a task from the queue, completes it and obtains another task. There is a task generator that generates all the tasks to be done and puts them into the queue. Each worker is always busy as long as there are tasks left in the queue.

The worker/task model is desirable if all the tasks are independent identities, i.e., any conventional "sequential" programs that do not have interactions with others. In our case, each coroutine is a task that synchronizes and communicates with many other coroutines. Therefore, a new task in the queue may be a previously suspended task that has a history associated with it. As a result, the task allocation can no longer be done transparently to the workers and extra effort is necessary to obtain the history of a given task and to check if the history can be restored under the local condition of a particular worker. It may be interesting to implement the CSP system based on this model and compare with the static one.

4.5.2 Distributed Port Table

Another interesting point concerns the location of the global port table. Currently it resides on a single node. The disadvantage of this central table scheme is that it may lead to contention. A port table that is distributed across many nodes could avoid this problem, but the difficulty in maintaining and locating desired entries is higher.

Chapter 5

Tuning the Algorithm for Better Performance

5.1 Factors and Metrics

In this chapter we discuss the performance of the algorithm described in Chapter 2 and its implementation described in Chapter 4. The time that a process spends in a specific alternative operation is affected by many factors. For example, the number of alternatives, i.e., guards, affects the amount of time required to scan the guard list. The amount of computation that a process conducts between two consecutive alternative operations also influences performance. Intuitively, the more frequently processes enter the alternative operation, the more likely collisions are to take place, potentially increasing the number of aborted operations.

The sleeping period, i.e., the amount of time a process waits after an alternative operation aborts, is the only parameter of the algorithm that has yet to be specified. Chapter 2 contained some *qualitative* discussion as to how this period should be set in order to avoid adverse situations. In particular, a very short sleep period could lead to an improbable but theoretically possible scenario which invalidates the liveness of the algorithm. The argument against a short sleep period is further reinforced from a performance standpoint, because it may cause a process to wake up when its neighbors are still in the ALT state, leading to additional aborted attempts. other hand, a sleeping period too long could lead to an unnecessarily long time spent in the SLEEPING state. Techniques must be devised to determine good values for the sleeping period under different conditions.

5.2 Test programs

A set of parameterized programs, representing different workloads, was designed for collection of statistics. The parameters are

topology The channel connection pattern among the processes. This determines the size of guard list in the alternative operation.

"interval" computation The amount of time each process spent in the RUNNING state between consecutive alternative operations. Each process executes the alternative operation in a repetitive loop.

sleep period The time a process spends in the SLEEPING state when it aborts.

One may argue that the size of messages exchanged between pairs of processes should also be considered. It is not considered here because the communication takes place *after* the rendezvous is reached, so it is in fact not part of the alternative operation. Also, in figure 2.4 the state is set to RUNNING before the communication routine is called, suggesting that communication is part of the interval computation.

Each of the test programs simulates a lattice of 16 communicating processes. Each process communicates with some number of its neighboring processes(referred to as the *degree*. Figure 5.1 shows the connection patterns of the meshes with degrees 4, 8, 10, and 12 respectively. The actual set of programs contains meshes with degree 4, 6, 8, 10, 12, and a full connection pattern, i.e., degree 15. Wrap around connections are used at the edges at the mesh to maintain symmetry. Each of the 16 processes repetitively executes an alternative operation, attempting rendezvous with one of its neighbors. When executed on the Butterfly, 16 processors are used so that each element of the mesh is created and run on a separate processor. The reason for this arrangement is that the coroutine implementation of the alternative algorithm guarantees that two neighboring CSP processes would never see each other in the ALT state, thereby reducing the possibility of abortion and

retry. Therefore, this arrangement is to analyze the *worst case* behavior of the alternative operation under different parameter settings.

The mesh connection pattern is chosen because it is *symmetric*, so that all the processes in a particular program have the same number of I/O guards listed in their respective alternative operations, conforming to the parameterized model. The symmetry of this topology also avoids *bottlenecks* that can influence the reliability of the result. Moreover, the mesh is full of cycles that can lead to deadlocks if the system is implemented incorrectly. which provides a challenging environment for testing the implementation. Appendix A shows the C code for the degree 4 program.

Each of the three parameters is independent of the others. The length of interval computation is chosen from an exponential distribution with a mean ranging from 500 microseconds to 16 milliseconds. The sleep period ranges from 100 microseconds to 12.8 milliseconds. The interested measures from those experiments are (1) the average transaction time. and (2) the abortion rate, i.e., the average number of abortions per transaction.

The two measures are calculated mean values from a normally distributed random space, each sample taken from recordings of individual transactions. The result shows that, in all of the topologies, with 100 repetitions of the alternative operation executed by each CSP process, the 95 percent confidence interval about the calculated mean value is less than 4 percent. In other words, the probability that the true mean value is within 4 percent of the calculated mean is 0.95. Therefore, 100 repetitions are sufficient.

It should be noted that all of the time measurements are derived from recordings of the real time clocks which have a resolution of 62.5 microseconds. The recordings include the operating system overhead, contain some truncation error. The truncation error, in theory, is 31.25 microseconds for each measurement. Therefore.

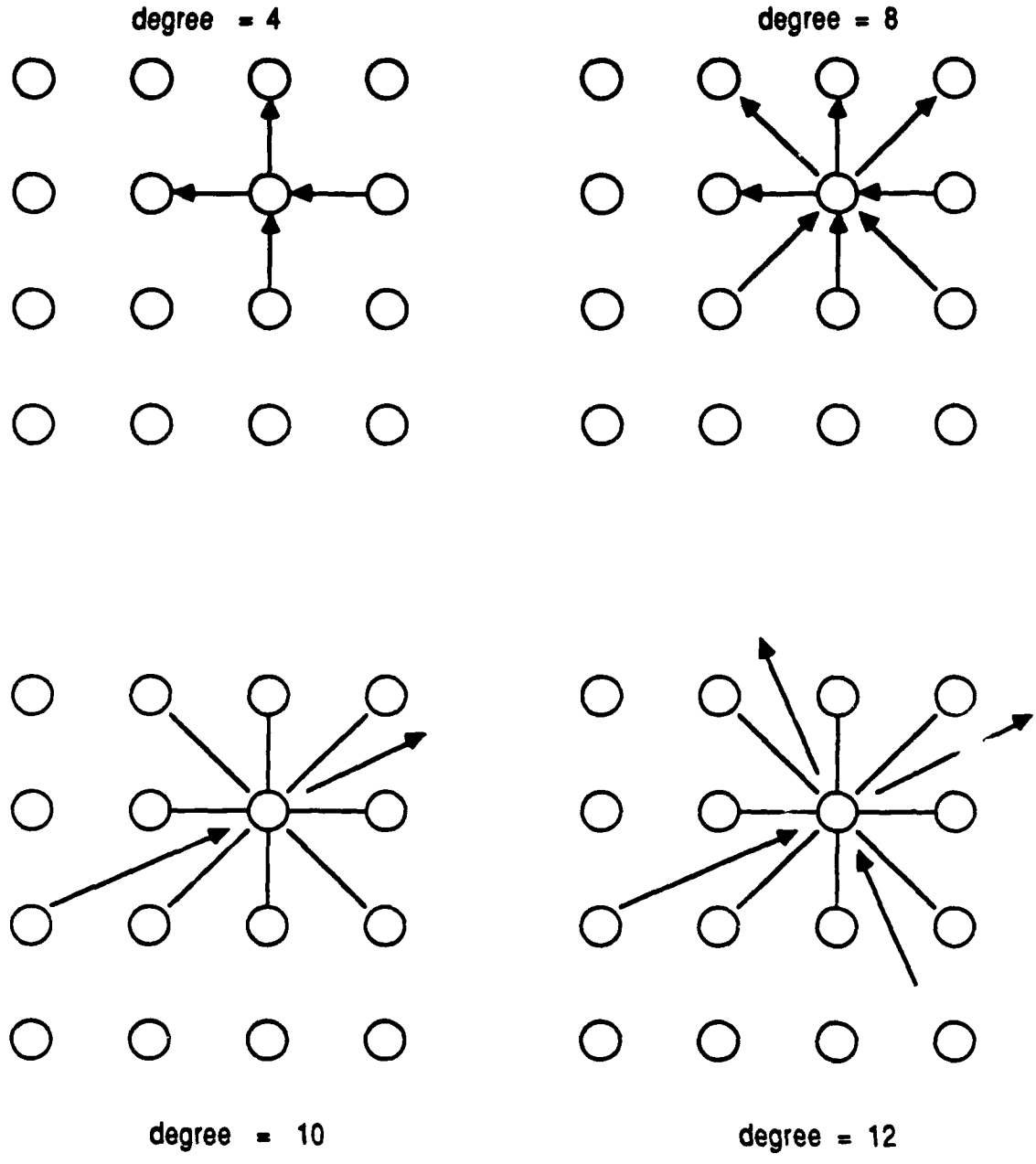

Figure 5.1: The Connection Patterns Of Various Degreed Meshes

for large values, e.g., several milliseconds, the truncation error can be ignored. The overhead is approximately 5 percent of the computation time and does not affect the *relative* relationships that are described later, nor the results that follow.

5.3 Results

Figure 5.2 shows the abortion rates when the interval computation is zero. The zero interval computation presents the worst case situation because any computation reduces abortion as mentioned earlier. It is observed that in all of the topologies, longer sleeping periods lead to smaller abortion rates. Intuitively, longer sleeping periods reduce the possibility that processes see each other in the ALT state, thereby reducing the probability of further abortions. Note in topologies with small degrees, e.g., 4 and 6, the abortion rate is not affected as dramatically as it is for large degree topologies as the sleeping period is varied. This is because the probability of seeing each other in the ALT state is already small for topologies with small degree.

Figure 5.3 shows the mean transaction times with zero interval computation. It is noted that each topology has a "best" sleeping period. This best period increases with the degree of the topology. When the list of guards becomes longer, the alternative algorithm tends to spend more time scanning the list, and therefore spends more time in the ALT state. As a result, a longer sleeping period is needed to avoid excessive abortions. However, an excessively long period is undesirable as well.

Table 5.1 shows the approximate "best" sleeping periods in each topology in milliseconds, along with the average abortion rate. Those values are approximate because the curves are drawn from only a few sample points. Also, the percentage in parentheses indicate the standard deviations of the respective mean values. They suggest that topologies with large degree have very large variances.

Table 5.2 shows the average time that each transaction spends in the SLEEPING

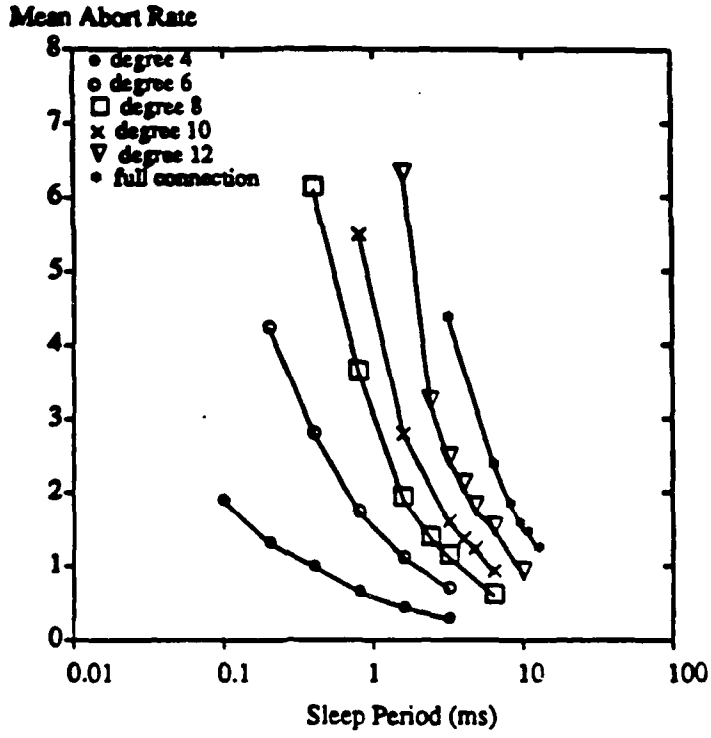

Figure 5.2: Abortion Rates under No Interval Computation

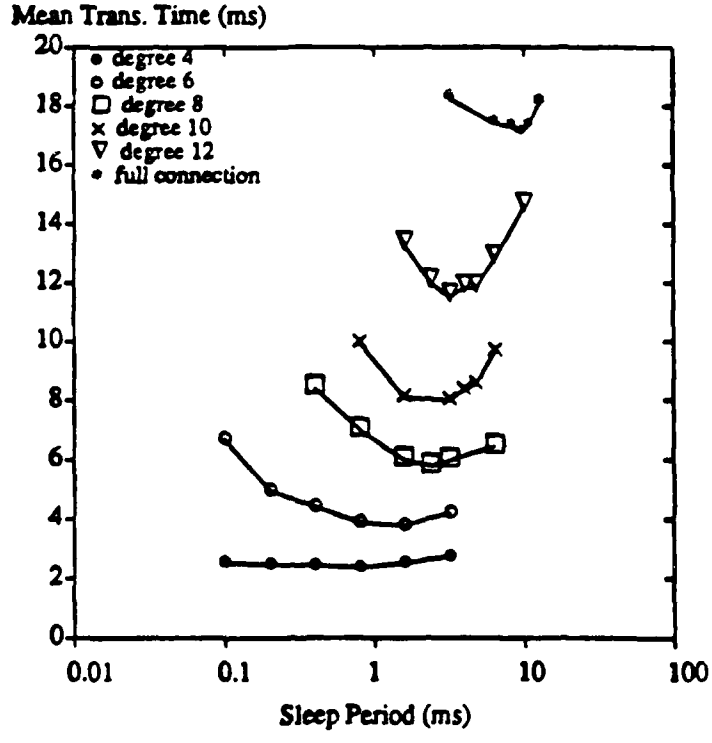

Figure 5.3: Transaction Times under No Interval Computation

Table 5.1: "best" Sleeping Periods under No Interval Computation

Degree	Avg. Abort Rate	Avg. Tran. Time	Sleep Period
4	(10%) 0.9	(25%) 2.0	0.6
6	(8%) 1.0	(58%) 3.3	1.2
8	(6%) 1.6	(67%) 5.1	2.4
10	(6%) 1.7	(75%) 7.2	3.2
12	(4%) 1.8	(78%) 10.4	4.8
fc	(5%) 1.5	(92%) 17.2	9.6

state for the "best" selection sleep periods shown in figure 5.1. The average time is calculated by multiplying average abortion rates by the sleeping periods. These figures are not precise due to the large variance. However, the tendency is clear; topologies with larger degrees spend a much larger portion of their time in the SLEEPING state.

Table 5.3 shows the proportion of time transactions spend in activities other than sleeping. The major activities are (1) the *Lock* operation, in the alternative algorithm (figure 2.4) before entering the WAITING state, and in procedure *CheckAndCommit*(figure 2.2), (2) the *CheckGuard* operation described in figure 2.2, (3) the busy wait loop used to scan the remote process if it is in the ALT state, and (4) the time sent in the WAITING state, if the process does not rendezvous during scanning. Other parts of the alternative algorithm are irrelevant to the interactions between the processes and do not spend much time in them. All numbers are in milliseconds. No time is spent in the *Communicate* procedure in this experiment because no data is transferred.

It is seen that the time spent in *CheckGuard* increases as the degree becomes larger because the guard list is longer. The time in the busy wait loop increases because processes spend more time in ALT. Similarly, the processes spend more time waiting for a signal in the WAITING state when there are more conflicts in high degree topologies. The time measurements of the *Lock* operation appear not

Table 5.2: Percentage Spent In the SLEEPING state

Degree	Ab. Rate ×Slp Prd	Avg. Trans. Time	Percentage
4	0.48	2.0	24%
6	1.08	3.3	33%
8	3.20	5.1	62%
10	4.76	7.2	66%
12	8.64	10.4	83%
fc	14.4	17.2	84%

Table 5.3: Time Spent Other Than Sleeping

Degree	Locking	CheckGuard	BusyWait	WAITING
4	0.137	0.077	0.33	1.31
6	0.141	0.093	0.46	1.35
8	0.143	0.110	0.66	1.49
10	0.148	0.119	0.79	1.60
12	0.136	0.145	0.97	1.82
fc	0.135	0.154	0.91	1.80

is not as closely related to the degree of the topology.

Figure 5.4 shows the average transaction time when there are different amounts of interval computation and the sleep period is set to the "best" value for each topology. The length of interval computation shown in the figure is the mean of an exponential distribution. Intuitively, the interval computation spreads alternative operations attempts by processes over a longer period of time, thereby reducing the frequency of conflicts. Since its purpose is the same as the sleep period, a similar relationship between the interval computation and the performance of the algorithm is expected. In particular, in order for the algorithm to achieve a performance close to the optimum, i.e., no abortion enforced, a longer interval computation is needed for topologies with larger degrees. This behavior suggests that the more sparsely an application executes the alternative operation, the closer the performance of each operation is to the optimum.

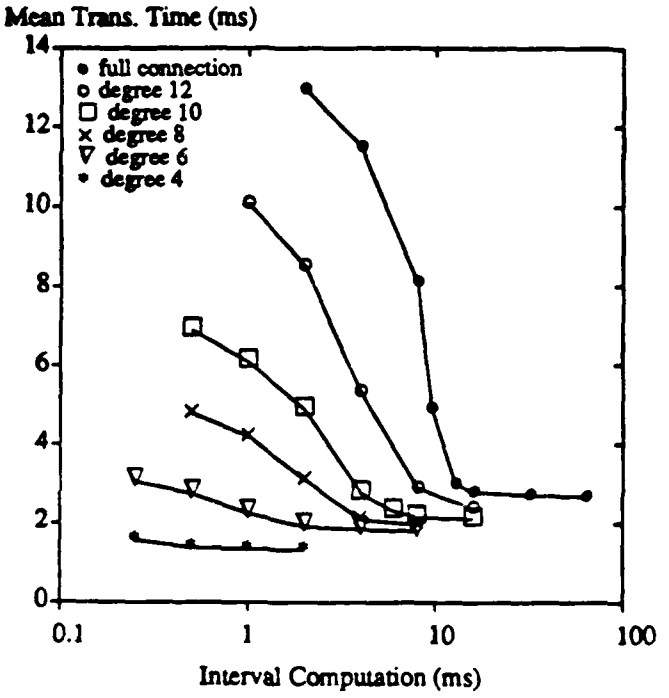

Figure 5.4: Transaction Times with Interval Computation

It is interesting to note that the least amount of interval computation necessary to achieve optimal performance, i.e., zero abortion, is comparable to the "best" sleep period in the fixed sleep period scheme. This coincidence is expected because these two quantities are handled the same way in the alternative algorithm (the RUNNING state and the SLEEPING state) and therefore should have the same influence on the performance. If the interval computation is too long, however, processes will be spending the extra time in the WAITING state waiting for their communicants to execute the alternative operation. This factor depends totally on the application program and cannot controlled by any alternative algorithm. The sleep period, on the other hand, must be optimized.

5.4 An Adaptive Approach To Setting The Sleeping Period

The result of previous section suggests that a sleeping period set to favor a particular configuration may induce undesirable performance under other circum-

stances. Unfortunately, one can not expect uniform communication patterns like those in the test programs in many situations. Furthermore, the large deviation from the mean transaction time implies that even in large degree and symmetric topologies a fix sleeping period may be excessive in many transactions.

To accommodate these circumstances, an *adaptive* approach of setting the sleeping period is necessary. In particular, one measure of whether the sleeping period should be reset is the number of abortions the current transaction has committed so far. Intuitively, any additional abortion in a given transaction may mean the current sleeping period is not long enough. This gives rise to the following scheme that dynamically sets the sleeping period.

The sleeping period is initially set to a small value on each transaction. During the lifetime of this transaction, the occurrence of any additional abortion is an indication of insufficient sleeping period. Therefore, the period is doubled, i.e., the process sleeps for a time twice as long as it did last time when it aborted. This scheme should adapt to the particular circumstance of an alternative operation quickly. It is clear that a large degree topology may take several abortions before it accumulates a suitable period, thereby introducing some overhead. However, some alternative operations that could have been completed using shorter sleeping period will not incur further abortions, and if they did need longer sleeping periods, the exponential approximation should quickly converge to the correct value. Table 5.4 validates this argument by showing the performance of such an adaptive scheme with the sleeping period set to 400 microseconds in the beginning of each transaction. Note that in topologies with large degrees the improvement is apparent because many transactions now spend much less time sleeping, while in small degree topologies the spared time is compromised by the overhead of additional abortion.

Table 5.5 shows the participation in the activities other than SLEEPINGin the

Table 5.4: Performance of the Adaptive Scheme

degree	abortion/tran	time(ms)/trans
4	0.3	1.8
6	1.6	3.3
8	1.9	4.8
10	2.5	6.9
12	2.9	9.7
fc	3.5	12.9

Table 5.5: Time Spent Other Than Sleeping for Adaptive Scheme

Degree	Lock	CheckGuard	BusyWait	WAITING
4	0.100	0.069	0.23	1.45
6	0.164	0.099	0.57	1.26
8	0.158	0.115	0.80	1.33
10	0.166	0.130	0.96	1.39
12	0.170	0.155	1.07	1.41
fc	0.173	0.169	1.23	1.35

adaptive algorithm.

The performance of the adaptive algorithm when there is interval computation is very close to what was shown in figure 5.4. The reason is that as the algorithm seldom aborts, so it is not very important how the sleep period is set.

5.5 Conclusion

In this chapter, the performance of the implementation of the alternative algorithm is investigated. It is observed that in a symmetric topology an alternative operation with a longer guard list to scan needs a longer sleep period to achieve better performance. However, an adaptive scheme is found to be more suitable than the fixed sleep period approach, and is expected to perform well in highly irregular topologies. Also, the experiment result suggests that application programs would suffer serious performance degradation if the alternative operation is executed repetitively with very little interval computation.

Chapter 6

Conclusion

This chapter summarizes the thesis work reported here as well as the possible extension.

6.1 Thesis Work Summary

In this thesis, a shared memory implementation scheme of the generalized alternative algorithm, which is of principle interest in CSP, is proposed. A CSP process executing the algorithm exercises an abort/retry protocol in cooperation with other processes to achieve rendezvous with exactly one of them, and exchange messages with it.

The correctness of this algorithm is proven based on some *safety* and *liveness* properties which are generally required by many distributed computing paradigms. The properties guarantee that an application program, will never experience deadlocks due to flawed internal operations of the algorithm.

The scheme has been realized and tested on a Butterfly Parallel Processor^{TM}. The features of the implementation includes the following:

- The C language is used for programming applications, and a library of procedures and macros is used for invoking CSP primitives.
- The port model is employed to provide communication capability (through alternative construct) to the application programs.

- An application can run on any number of processors with the constituent processes statically allocated to one of the available processors.

A set of programs is devised to test the implementation, as well as to produce some statistics to help tune the system for better performance. In particular, the "sleeping" period during which a process should stay aborted, in the context of the abort/retry mechanism, is found to have major impact on performance. An adaptive method is devised and proves satisfactory in dynamically setting the sleeping period suitable for a wide range of workload configurations facing the alternative algorithm.

6.2 Possible Extensions

There are plenty of rooms for optimization in this rudimentary, albeit legitimate, implementation before it can be put into practice.

From the application programmer's point of view, the interface to the system can be made much more friendly. For example, the current implementation is built on top of an existing language through procedure calls. Therefore Only a limited amount of compile-time checks are possible. It is desirable that a preprocessor be introduced which does all of the static checking at compile time and generates information invisible to the users but necessary to the "back end" system described in this thesis. The interface can be made more friendly by providing a syntax that is more concise and rigorous, avoiding the awkwardness of procedure calls.

Other issues that have yet to be explored include implementation of strategies for dynamic load balancing which migrates tasks among processor nodes according to some heuristics and run time information, or alternatively, according to explicit user control.

Bibliography

[1] A. A. Aaby and K. T. Narayana.
A Distributed Implementation Scheme For Communicating Processes.
ICPP Proceeding, 942–949, August 1986.

[2] B. Thomas, et al.
Butterfly Parallel Processor Overview.
BBN Report No. 6148, BBN Laboratories Incorporated, March 1986.

[3] R. Bagrodia.
A Distributed Algorithm To Implement The Generalized Alternative Command In CSP.
The 6th International Conference On Distributed Computing Systems, 422–, May 1986.

[4] M. Beeler.
Butterfly Parallel Processor Tutorial (for the C language).
BBN, November 1985.

[5] A. J. Bernstein.
Output Guards and Nondeterminism in 'Communicating Sequential Processes'.
ACM Transactions on Programming Language and Systems, (2):234–238. April 1980.

[6] *Chrysalis Programmer's Manual.*
BBN, December 1985.

[7] G. N. Buckley and A. Silberschatz.
An Efficient Implementation for the Generalized Input-Output Construct of CSP.
ACM TOPLAS, 5(2):223–235, April 1983.

[8] M. Collado, R. Morales, and J. J. Moreno.
A Modula-2 Implementation Of CSP.
ACM SIGPLAN Notices, 22(6):25–, June 1987.

[9] E. W. Dijkstra.
Guarded Command, Nondeterminism and Formal Derivation of Programs.
CACM, (8):453–457, August 1975.

[10] E.W. Dijkstra, W.H. Feijen, and A.J.M. van Gasteren.
Derivation of a Termination Detection Algorithm for Distributed Computations.
Information Processing Letters, 16(5):217–219, june 1983.

[11] R. E. Filman and D. P. Friedman.
Communicating Sequential Processes, chapter 10.
Computer Science Series, 1984.

[12] G. A. Ford and R. S. Wiener.
Modula-2, A Software Development Approach.
John Wiley And Sons, NY, 1985.

[13] N. Francez.
Distributed Termination.
ACM Transactions on Programming Language and Systems, 2(1):42-55, 1980.

[14] N. Francez.
Fairness.
Springer-Verlag, New York, 1986.

[15] G. F. Pfister, et al.
The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.
Proceeding of the 1985 International Conference On Parallel Processing, 764-771, August 1985.

[16] D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh.
Cedar.
COMPCON-84, IEEE Computer Society Conference, 306-309, February 1984.

[17] C. A. R. Hoare.
Communicating Sequential Processes.
Computer Science, Prentice Hall, 1985.

[18] C. A. R. Hoare.
Communicating Sequential Processes.
CACM, (8):666-677, August 1978.

[19] R.V. Hogg and A.T. Craig.
chapter 6.
Macmillan Publishing Inc., 1978.

[20] I.B. Barron, et al.
The Transputer.
Electronics, 109, November 1983.

[21] *OCCAM Programming Manual.*
Inmos Ltd., 1982.

[22] D. R. Jefferson.
Virtual Time.
ACM Transactions on Programming Languages and Systems, (3):404-425, July 1985.

[23] R. A. Karp.
Proving Failure-Free Properties Of Concurrent Systems Using Temporal Logic.
ACM Transactions on Programming Language and Systems, 6(2):239-, April 1984.

[24] R. B. Kieburtz and A. Silberschatz.
Comments on 'Communicating Sequential Processes'.
ACM Transactions on Programming Language and Systems, (2):218-225, Oct. 1979.

[25] *MC68020 32-Bit Microprocessor User's Manual.*
Motorola Inc., 1984.
Printed by Prentice-Hall Inc.

[26] J. Misra.
Distributed-Discrete Event Simulation.
ACM Computing Surveys, (1):39-65, March 1986.

[27] B. Moxon.
The Butterfly RAMFile System.

BBN Report No. 6351, BBN Advanced Computers Incorporated, 1986.

[28] S. Owicki and L. Lamport.
Proving Liveness Properties Of Concurrent Programs.
ACM Transactions on Programming Language and Systems, 4(3):455–495, July 1982.

[29] G. F. Pfister and V. A. Norton.
"Hot Spot" Contention and Combining in Multistage Interconnection Networks.
IEEE Transactions on Computers, C-34(10):943–948, October 1985.

[30] R. Pountain.
The Transputer and its Special Language OCCAM.
BYTE, 361–366, August 1984.

[31] D. A. Reed and R. M. Fujimoto.
Multicomputer Networks: Message-Based Parallel Processing.
Computer Science, MIT Press, 1987.

[32] D. A. Reed, A. D. Malony, and B. D. McCredie.
Parallel Discrete Event Simulation: A Shared Memory Approach.
Proceedings of the 1987 ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems, 15(1):36–38, May 1987.

[33] J.S. Schwarz.
Distributed Synchronization of Communicating Sequential Processes.
Technical Report, Dept. of A.I., Univ. of Edinburgh, Scotlnad, 1978.

[34] Z. Sun and X. Li.
CSM: A Distributed Programming Language.
IEEE Transactions On Software Engineering, SE-13(4):497–, April 1987.

[35] *The Uniform System Approach to Programming the Butterfly Parallel Processor.*
BBN Lab. Inc., November 1985.

[36] J.L.A. Van De Snepscheut.
Synchronous Communication Between Asynchronous Components.
Infomation Processing Letters, 13(3):127–130, 1981.

[37] N. Wirth.
Programming In Modula-2.
Springer-Verlag, Heidelberg, 1982.

[38] D. Zobel.
Transformations For Communication Fairness In CSP.
Information Processing Letters, 25:195–198, May 1987.